KU LEUVEN

FACULTEIT INDUSTRIËLE
INGENIEURSWETENSCHAPPEN

TECHNOLOGIECAMPUS GENT

Combating address-sensitive behaviour in MVEEs

A study of memory layout dependent program state and control flow

Adriaan JACOBS

Promotor: prof. Stijn Volckaert

Co-promotors: Ruben Mechelinck
Jonas Vinck
Alexios Voulimeneas

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Elektronica-ICT Internet Computing

Academiejaar 2020 - 2021

Acknowledgements

Ik heb nog nooit zoveel geleerd in een jaar dan ik gedaan heb in dit laatste jaar. Dat heb ik in de eerste plaats te danken aan mijn promotor, prof. Stijn Volckaert, en mijn co-promotoren Alex, Jonas en Ruben. Bedankt om mij in korte tijd te voorzien van dit onderwerp en voor de brede en intensieve ondersteuning tijdens het onderzoek. Also to you Alex, in English: thank you for the past year.

Naast het werk op zich, besluit deze thesis ook mijn Masteropleiding tot Industrieel Ingenieur. Op de talloze dagen waarop ik het moeilijk had om door te zetten met studeren en te gaan voor mijn examens, kon ik steeds, zonder fout, rekenen op de onwrikbare steun van mijn ouders. Het is in de eerste plaats hun verdienste dat ik de lat hoog legde in de afgelopen jaren, en nu de eindmeet in zicht is ben ik hen daar buitengewoon erkentelijk voor. Hoed af, mama en papa. Bedankt.

Dan heb ik ook nog mijn broer Quinten te bedanken. Telkens ik niet weet waar ik het doorzettingsvermogen vandaan moet halen, is het een ongeloofelijke luxe om naast mij te kunnen kijken en jouw spoor te volgen. Bedankt om al meer dan 20 jaar de weg te leiden.

Daarnaast wil ik ook nog Alicia bedanken voor haar doorgedreven *solo-carry to global* tijdens de vele labo's die we samen hebben uitgezeten. Bedankt daarvoor, en voor alle andere momenten waarop ik kon rekenen op jouw steun het afgelopen anderhalf jaar.

Vervolgens wil ik vrienden en mede-studenten bedanken voor de plezante momenten de afgelopen jaren. Bedankt Sander, Sander, Wander, Victor, Bert, Jef, Mathieu, Elliot, Eline, Margot, Sophie en Cedric.

Ten slotte wil ik Yuri Cauwerts bedanken om mij op zijn enthousiaste manier te introduceren tot C en Linux, en Dries Deschout om het geduld te hebben om op al mijn vragen over Rust, C++, dagactiviteiten in België tijdens Corona, en buildsystemen te antwoorden. Het lijkt erop dat al deze dingen behalve één een groot deel zullen uitmaken van mijn tijdsbesteding de komende jaren.

Samenvatting

Multi-variante Uitvoeromgevingen (MVUO's) beloven uitgebreide bescherming tegen aanvallen op basis van geheugenfouten door meerdere gediversifieerde varianten van hetzelfde programma parallel uit te voeren, waarbij de toestand van het programma op bepaalde Rendez-Vous Punten (RVPs) wordt gecontroleerd en vergeleken. Bestaande programma's kunnen zich echter van nature niet-deterministisch gedragen wanneer ze worden onderworpen aan bepaalde vormen van diversificatie die een typische MVUO of besturingssysteem toepast. Een opmerkelijk probleem is Adres-Gevoelig Gedrag (AGG), waarbij specifieke adreswaarden de *control flow* van een programma beïnvloeden of de staat ervan op het moment van de RVP's aanpassen, wat goedaardige divergenties in de MVUOs veroorzaakt. Hoewel dit de toepasbaarheid van MVUOs in de praktijk beperkt, bestaat er geen eerder werk dat de volledige reikwijdte van dit probleem verkent. Dit werk beoogt die leemte op te vullen en onderzoekt de verschillende manieren waarop AGG wordt uitgeoefend door programma's in de praktijk.
We presenteren een overzicht en categorisatie van het probleem om de oorzaken en gevolgen ervan te belichten, alsmede om een theoretisch referentiekader te scheppen. Bovendien onderzoeken we in detail de mogelijkheid om het AGG van een programma automatisch te verwijderen of te neutraliseren met behoud van maximale diversificatie. We stellen verschillende strategieën voor die we evalueren op basis van criteria zoals doeltreffendheid, veiligheidsimpact en prestatie-overhead. Sommige van de strategieën in dit werk kunnen alle door ons onderzochte voorbeelden van AGG op transparante wijze onderdrukken.

Trefwoorden: Niet-determinisme, Multi-Variante Uitvoeringsomgevingen, Geheugenlayoutdiversificatie, Adresgevoelig Gedrag

Abstract

Multi-Variant Execution Environments (MVEEs) promise comprehensive protection against memory error exploitation attacks by running multiple diversified variants of the same program in parallel, monitoring and comparing program state at certain Rendez-Vous Points (RVPs). However, real-world programs can naturally behave non-deterministically when subjected to some of the diversification a typical MVEE or Operating System (OS) employs. A notable issue is Address-Sensitive Behaviour (ASB), where specific address values influence the control flow of a program or taint its state at the RVPs, triggering benign divergences in MVEEs. Although this limits the applicability of MVEEs in practice, no prior work exists that explores the full scope of this problem. This thesis aims to fill that gap and to investigate the various ways in which ASB is exercised by real-world programs. We present an overview and categorization of the problem to highlight its causes and consequences, as well as to define a theoretical frame of reference. Additionally, we examine the possibility of automatically removing or neutralizing the ASB of a program while retaining maximal diversification. We propose and analyze several strategies and evaluate them on metrics like efficacy, security impact and performance overhead. Some of the strategies we provide are able to transparently mitigate all the examples of ASB that we surveyed.

Keywords: Non-determinism, Multi-Variant Execution Environments, Memory Layout Diversification, Address-Sensitive Behaviour

Contents

List of Figures

Acronyms

API Application Programming Interface. 8, 10, 22, 23, 27, 54, 55, 65

ASB Address-Sensitive Behaviour. v, vi, viii, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42, 43, 48, 58, 59, 60, 61, 62, 63, 68, 72, 74, 75, 77, 78, 79

ASLR Address Space Layout Randomization. 3, 4, 5, 6, 18, 64

CP-MON Cross-Process Monitor. 6

CPU Central Processing Unit. 9, 26

DCL Disjoint Code Layouts. 13, 35

ELF Executable and Linkable Format. 18

FPU Floating Point Unit. vi, 88, 89, 90, 91

GHUMVEE Ghent University Multi-Variant Execution Engine. 3, 6, 11, 13, 24, 59, 62, 63, 64, 65, 71, 72, 75, 79

IPC Inter-Process Communication. 38

IP-MON In-Process Monitor. 3, 6

IR Intermediate Representation. 20, 22, 64, 69, 75, 78

MLD Memory Layout Diversification. v, 3, 4, 5, 9, 16, 17, 32, 35, 38, 39, 59, 62, 63, 65, 79

MSVC Microsoft Visual C++. 11

MVEE Multi-Variant Execution Environment. v, 3, 4, 5, 6, 8, 9, 12, 13, 14, 15, 16, 23, 24, 31, 32, 34, 35, 37, 39, 43, 58, 59, 61, 62, 66, 77, 79

OS Operating System. v, 10, 18, 19, 48, 62

RAM Random Access Memory. 72

ReMon Relaxed Monitor. 3, 6

ROP Return-Oriented Programming. 5, 58, 79

RVP Rendez-Vous Point. v, 3, 4

SIMD Single Instruction Multiple Data. 26

STL Standard Template Library. 39, 70

TOCTOU Time-Of-Check To Time-Of-Use. 13

UAS Unified Address Space. viii, 44, 45, 46, 47, 50, 51, 52

VSAS Variant-Specific Address Space. viii, 44, 45, 46, 47, 51, 52

Chapter 1

Introduction

Today's most widely-used low-level programming languages (C and C++) are fundamentally memory-unsafe because of their lack of enforced bounds-checking or restrictions on memory accesses, opening up applications to a wide array of memory-related exploitation attacks, such as buffer overflows and use-after-free. A promising mitigation strategy in this space is the use of Multi-Variant Execution Environments (MVEEs) where N variants of the same program run in lockstep, supervised by a monitor that feeds them the same input and terminates execution should they diverge from each other. The variants are generated from the same program using software diversity techniques, making it hard for an attack to have the same effect in all of them at the same time.
The state-of-the-art in these systems is ReMon [1], which consists of the *ptrace*-based cross-process monitor GHUMVEE [4] and a highly efficient in-process monitor (IP-MON). ReMon intercepts variants at every system call[1] and compares their arguments to determine if they have diverged. This is referred to as a monitor with system call granularity. In general, the points at which MVEEs intercept variants are called Rendez-Vous Points (RVPs).

Some of the software diversity techniques amplified by Multi-Variant Execution Environments (MVEEs) apply Memory Layout Diversification (MLD) to the variants. These techniques include, among others, randomizing memory allocators [5, 6], reversing the direction of the stack [7, 8] and Address Space Layout Randomization (ASLR) [9]. Key here is to not just provide a merely randomized layout for the respective processes' stack and heap segments, but rather to ensure that equivalent code segments are guaranteed not to overlap when comparing the variants' address spaces [10], since that ensures that gadgets (small pieces of executable code that can be chained together to perform arbitrary executions [11]) cannot be reliably jumped to in all variants at the same time. Meanwhile, previous research has shown that the execution of programs is not consistent in the face of changing memory layouts [4]: they will produce different results, follow different code paths or have different side-effects depending on where the memory they use actually resides in the address space. This is called the Address-Sensitive Behaviour (ASB) of a program, and since MVEEs consider variants that do not execute the same system calls with equivalent arguments to be malicious, this causes a lot of real-world software to trigger benign divergences and currently contributes to the limited applicability of MVEEs in

[1] Some system calls are considered security-sensitive and handled by GHUMVEE while the others are forwarded to IP-MON.

practice.

Apart from Volckaert et al., few researchers have reported on this problem [4] and there has been no previously published work detailing the scope of it. In this thesis, we therefore make an effort to provide a first complete overview of ASB. In addition, we present failing and promising techniques for both detection and mitigation in the hope of moving the MVEE community a step closer towards a general solution for address sensitivity in the context of security-oriented MVEEs and to serve as a background for future research.

We aim to provide a flexible design that supports as much diversification as possible, including but not limited to ASLR and the use of different allocation algorithms in the variants. Variant generation processes and the MLD they employ evolve continuously, driven by new research insights. Address-sensitivity remains prevalent as long as any memory layout diversification is applied though, so it is important that a solution is able to adapt to current and future methods.

The content of this work is laid out as follows: in Chapter 2, we provide some background information about the concepts that are important to understand the rest of this work. Chapter 3 provides an overview of address-sensitive behaviour detailing the various ways in which it can appear and categorizes it based on its causes and its effects at the RVP-level. In Chapter 4 we present a series of approaches for the detection of ASB at its introduction into the program and the automatic mitigation of its effects, as well as a reflection on their shortcomings, specifically the security impact they entail. After that, Chapter 5 lays out the implementation details of an approach that was tested in practice to motivate some properties that constrain/optimize its performance and then in chapter 6 we critically evaluate the prototype implementation on metrics such execution speed and ASB coverage. We also benchmark its transparency on the GNU core utilities. Finally, Chapter 7 concludes this work and summarizes the key insights and contributions of thesis. Additionally, we make a look to the future and propose some adaptations of the current system to improve its performance on the different metrics.

The direct proceedings of this thesis outside of the main text constitute a set of patches and passes for the Clang compiler [12] and an interposer library that links into the target application. All of the code produced as part of this work is hosted on GitHub at `github.com/ku-leuven-msec/masterproef-20202021-adriaanjacobs`.

Chapter 2

Background

Memory errors such as buffer overflows and use-after-free bugs, ubiquitous in programs written in memory-unsafe languages such as C and C++ [13], can be exploited by interacting with the program in an unexpected way. This was famously first demonstrated on a large scale by the Morris worm in 1988 [14] and many more exploits as well as mitigation strategies have been presented ever since [15]. One such techniques is the use of *software diversity* to transform programs in a way that does not change their semantics[1] [16] such that it becomes harder for attackers to construct exploits based on the memory layout of a program[2], in the case of Memory Layout Diversification (MLD). Some of these techniques have seen widespread adoption, most notably the use of Address Space Layout Randomization (ASLR) on modern operating systems [9]. However, even though software diversity techniques raise the bar for attackers, they do not fundamentally prevent attacks from explicitly targeting the applied diversity and neutralizing it [10]. This is exemplified by the repeated bypassing of ASLR in recent years [17].

Multi-Variant Execution Environments (MVEEs) amplify software diversity techniques by running multiple diversified variants of the same program in parallel, monitoring their behaviour and providing the same input to all of them. If the monitor notices that the variants are executing different code, it considers this a *divergence* and terminates execution. Since all the variants are diversified differently but receive the same inputs, a successful attack needs to bypass the diversification of all variants at the same time without being able to interact with them individually. Volckaert et al. showed that an MVEE itself can additionally employ supplementary diversification that ensures that Return-Oriented Programming (ROP) attacks cause a divergence among the variants [10], protecting programs ran inside an MVEE from an important class of attack vectors.

The concept behind security-oriented MVEEs was originally introduced by in 2006 by Cox et al. [18] and has seen significant research interest over the years due to its promise of comprehensive security with low run-time overhead: because a lot of programs typically do not make use of the available parallelism on modern hardware [19], the overhead of running multiple variants at the same time is relatively small [1]. The current state-

[1] Provided it is free of errors.

[2] Note that software diversity is broader than merely memory layout diversification. However, we are primarily concerned with MLD in this work.

of-the-art is in MVEEs is ReMon, which intercepts the variants at every system call. Alternative granularities for monitoring have been proposed [2], but system call granularity has prevailed because it provides an attractive balance between performance and security: it is assumed that, to have any real influence on a system, an attack would eventually need to execute system calls. At the same time, they are executed infrequently enough that the overhead associated with intercepting and, in some cases, *syncing* variants on them is limited. To reduce performance overhead even further, ReMon uses an in-process monitor (IP-MON) that handles security-insensitive calls like `gettimeofday`, whereas a cross-process monitor (CP-MON) called GHUMVEE handles the other system calls, as illustrated by Figure 2.1[3]. However, the distinction between these is irrelevant for this work.

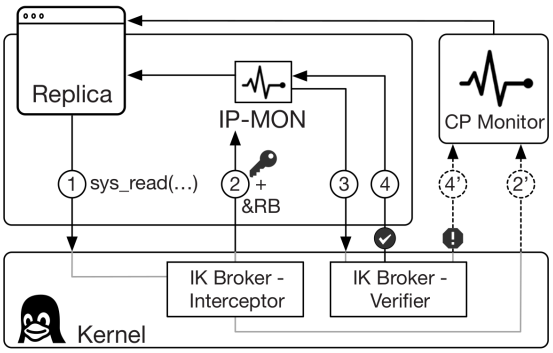


Figure 2.1: "ReMon's major components and interactions.", source: [1].

To verify if two system calls are equivalent, Salamat developed a formalization of what *equivalent* system calls are across the variants [19]. Non-pointer arguments must be equal and for pointer arguments, the allocation contents they refer to are compared instead. These allocation contents can either be an opaque buffer like in the case of the `write` system call, or a structured buffer such as the array of `iovec` **`struct`**s that `writev` takes. If any information about the layout of a buffer is known, the rules for argument equivalency are recursively applied. If not, the buffers must be equal byte-per-byte.

The exception that this formalization makes for pointer arguments already highlights the problem that this thesis tries to solve. Because of the diversity in the variants, introduced by ASLR but also techniques like randomized function frame padding [5] or the use of different dynamic allocation algorithms [6, 5], the pointer values that refer to *logically equivalent* objects and allocations are different in the variants. Hence, pointer arguments cannot simply be compared for equality to determine their equivalency.
However, the influence of the diversified memory layouts of the variants on the system calls they execute does not remain limited to these different pointer arguments. Because the order and structure of allocations is different in the memory of each variant, the contents of uninitialized data like padding bytes between **`struct`** members may also be different. When writing out **`struct`** contents to a pipe or file using `write`, the buffer contents will differ [20]. More trivially, if the **`struct`** has a pointer member in the first place, the buffer contents will also be different. To the monitor, it is opaque whether the content of these

[3]The word "replica" is also used to refer to a variant. For more information about the internal operation of ReMon, depicted in this figure, we refer to [1].

buffers is different as part of some kind of exploit or because of uninitialized data and it will terminate execution in the variants.
The influence of memory layout on the variants is additionally not limited to modifying only the arguments to the same system calls. The control flow of a program can be completely different based on which specific memory layout it faces. Volckaert et al. report on hash tables with pointers as keys that grow/rehash at different moments in the variants because the pointers, due to their pseudo-random value, are distributed differently over the buckets. Any iteration over such a hash table also accesses the pointers in a different order, potentially leading to entirely different code being executed in the variants. All of these cause *benign divergences*, making the monitor terminate execution without any malicious code being executed.
When the behaviour of a program depends on its memory layout as in the aforementioned cases, that is called Address-Sensitive Behaviour (ASB).

In the next chapter, we investigate these and more cases in more detail and propose categorizations for ASB.

Chapter 3

Address-sensitive behaviour

In this chapter we investigate the nature of address-sensitive behaviour in detail. First, we provide a series of examples of programming constructs that introduce ASB into a program. With this, we intend to help build intuition about the problem, as well as provide a benchmark to evaluate categorization, detection and mitigation efforts against. Secondly, we propose a definition of ASB, covering at least all examples provided in this chapter. Lastly, we discuss the categorization of ASB based on different commonalities among its causes and effects.
As a frame of reference for the following sections and rest of the thesis, we introduce some terminology and assumptions below:

1. What constitutes a divergence depends on the monitoring granularity of the MVEE. For this section and in the rest of this work we assume a system call granularity and define the equivalence of system call sequences according to the formal rules Salamat provides [19]. We then define any divergence as a non-equivalent system call sequence in the variants. This is in line with how some modern state-of-the-art MVEEs monitor execution: they assume that system calls are invoked relatively infrequently but are nonetheless required in order for an attack to have meaningful influence on a system [2]. This is an attractive balance of security and performance for most MVEEs.
   Secondly, since this work is concerned with removing false positives, we always use the term *divergence* to refer to *benign divergences*, unless otherwise specified.

2. *Logical allocations* are allocations that, irrespective of the address at which they reside, represent the same logical piece of memory that every variant operates on. They are allocated and deallocated in the variants from the same call sites and can represent both dynamically and statically allocated blocks. They correspond to the same *objects* in all variants, in the terminology of the C11 standard [21, Chapter 3].

3. To encourage generality, flexibility and applicability of potential mitigation strategies, we consider a hypothetical form of diversification in this analysis that makes every logical allocation in the program have a potentially different virtual address in the variants. Essentially, no assumptions can be made about the variant-specific address of a logical allocation in a program; neither that it is consistently the same nor that it is consistently different except in places where the Application Programming Interface (API) explicitly provides such constraints, e.g. `MAP_FIXED` [22] or

`aligned_alloc` [21, Section 7.22.3.1]. However, we assume that MLD does not include the use of different primitive type widths or **`struct`** layouts. More generally, we assume that logical allocations are equal in size in all variants and contain the same logical values at the same offsets. There have been MVEEs that diversify even these properties, most notably distributed MVEEs that run variants on different physical machines with different CPU architectures to further diversification [23, 24, 25].
We evaluate in later chapters to which end this maximal diversification can be supported while successfully neutralizing all ASB.

4. We do the analysis of the issue here for a C/C++-like language where memory addresses are explicitly modifiable or inspectable values in the source code. In languages with implicit references like Java, address-sensitive behaviour could definitely still occur, e.g. via the most common `Object.hashCode()` implementation [26], using an `Object` as key in `HashSets` or `HashMaps` etc., but we make no further effort to explicitly relate the concepts laid out in this work to those languages or paradigms.

5. We take concepts and terminology from a Linux environment and little-endian architecture. However, they should apply to most modern desktop operating systems that use a flat memory model and virtual memory management, potentially under a different name.

6. We assume that whenever memory addresses are introduced into a program, e.g. through an allocation call, they initially appear in source code with a pointer type, i.e. `T*` for `T` any type. This facilitates discussion: although it would be possible for an implementation to provide a `malloc` prototype that returns an instance of **`long`**, this would not be a standards-conforming implementation [21, Section 7.22.3.4] and it is not done by any major vendor. Note that this does not mean that we always assume every instance of pointer type to be a memory address, nor every memory address to be an instance of pointer type.

7. We interpret *(allowed) operations* broadly in the discussion below, covering all operations that most modern compilers such as Clang [12], GCC [27] and MSVC [28] support or can be made to support via compiler flags, unless otherwise specified. This includes many operations that are technically undefined, unspecified or implementation defined behaviour according to the C11 or C++17 standard, but since their use is widespread [29], there is little practical value contributed by a system that supports only strictly well-defined programs.

3.1 Examples of ASB that cause benign divergences

By no means is this an exhaustive list of every possible way in which a memory address can be used to alter system call sequence, but we cover a selection of the most prevalent cases here.

Pointer as key in hash table

A common way in which ASB manifests itself inside a code base is through the use of hash tables, where pointer values are used as keys.
The way a hash table works is that upon insertion a key is transformed into a different value through a hash function and then used to calculate an index into a contiguous array of buckets to find a bucket where the key is then placed. Commonly, this is done by applying a modulo operation using the hash value and the amount of buckets in the table, called its capacity. The bucket where a specific key resides can then be found in $O(1)$ time-complexity by applying the same process again to that key, which is the main advantage of a hash table. However, multiple keys might hash to the same value and the set of possible hashed values is typically significantly bigger than the amount of buckets present, so the bucket at the index that was calculated for a specific key might already contain multiple other entries. When that happens, those keys are stored next to each other inside the same bucket and upon retrieval the right key in the bucket still needs to be found after finding the right bucket for a specific key. If the amount of buckets is small, they will quickly contain relatively many keys and the look-up process will slow down. This is often mediated by monitoring the amount of keys being stored in the buckets and growing the hash table if a certain threshold is reached. Since the growing operation affects the capacity of the table, the positions of all entries need to be recomputed.
It is important to note that the API of this basic hash table does not provide any guarantees about the internal ordering of the keys by itself due to the typically hard-to-predict nature of the hash result. The only guarantee is that the same set of keys will produce the same internal ordering in the hash table every time. Therefore, when a hash table is used with pointers as keys, the most straightforward iteration over the buckets, i.e., according to their internal ordering, will pass the inserted pointers in a seemingly unrelated ordering compared to their insertion order. This is the first way a hash table can introduce ASB: if it is iterated over, different actions could be taken based on the pointer value that is assessed, and those actions might eventually cause a diverging system call pattern. The most straightforward example would be releasing the memory the pointer values refer to during the destruction of such a hash table, potentially causing pages to be unmapped in a different order, or not at all, in some variants.
The second way a hash table can cause ASB is through the condition that determines when the capacity should be increased: this will typically not only take into account just the total number of keys present, but also the average and maximum entries in a bucket or the number of non-empty buckets. Since the amount of keys in a bucket depends on the values of the keys themselves via the amount of collisions that happened, the decision to grow the table will inevitably be taken after a different amount of insertions in the variants. Since growing the table could end up requesting more memory from the OS via a system call this could again cause a divergence.

Checking the alignment of a memory address

Many applications check the alignment of a memory address to a certain boundary at some point using a variety of integer arithmetic. This leads to two distinct types of address-sensitive behaviour: one where pointer values are modified in an address-dependent way such that they do not refer to the same logical byte in the variants anymore, and another

where the alignment of a pointer value to certain boundary is part of a conditional evaluation that determines control flow in the variants. For example, Volckaert et al. report that *glibc*'s `ptmalloc` implementation requires that every new region it uses is aligned on a 1MB boundary. To achieve this, it `mmaps` a region of 2MB, finds a 1MB aligned pointer inside it and `munmaps` the unneeded pages above and below it, as shown in Figure 3.1.

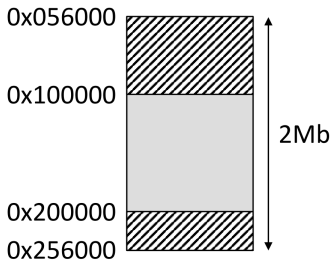


Figure 3.1: "Aligned allocation in ptmalloc", source: [2].

The 1MB aligned pointer value will be at a different offset into the 2MB region in all variants, because `mmap` only guarantees that it will return pages on a 4KB alignment [22]. Hence, one variant might get a region starting at one page before the next 1MB boundary, whereas another might get a page just after a 1MB aligned one, putting the next 1MB aligned pointer somewhere in the middle of the region. Therefore, the sizes of the regions that `ptmalloc` unmaps will be different in the variants, leading to a divergence. GHUMVEE implements a dedicated fake system call to support this [2].

Using relative pointers

Out of memory efficiency concerns, an application may not use a certain pointer variable to refer to an absolute memory location, but rather interpret it as the offset to another memory location. This can be both the location of the offset itself (called self-relative pointers) or some other pointer entirely. If the program can guarantee that this offset will be small enough to fit in a integer type smaller than the platforms pointer size, space can be saved. MSVC even supports an extension of the C++ language for this called *based pointers* [30].
According to the C++17 standard the behaviour of pointer subtraction is undefined when the pointers do not point to the same *array object* [31, Section 8.7.5]. However, on typical implementations, the result is equivalent to the difference of the integer representations of both pointers. Therefore, when computing offsets via pointer subtraction across allocations, a dependence on memory layout is introduced and when allocations are diversified this will give different results in the variants. This does not necessarily lead to a divergence and the resulting value will typically only be used to reconstruct some other pointer value somewhere else, but they do introduce another address dependent value into the program, usable for other ASB. This is exemplified by Listing 3.1.

```
std::vector<int32_t> rel_ptrs;
char* base_ptr = (char*) malloc(sizeof(some_struct));
size_t N = 10;
for (int i = 0; i < N; i++)
    rel_ptrs.push_back((char*)malloc(sizeof(some_struct))
                                                    - base_ptr);
std::sort(rel_ptrs.begin(), rel_ptrs.end());
std::vector<some_struct*> abs_ptrs;
for (auto rptr : rel_ptrs) {
    some_struct* abs_ptr = (some_struct*)(base_ptr + rptr);
    printf("abs_ptr:␣%p␣\n", abs_ptr);
    *abs_ptr = { /* data */ };
    abs_ptrs.push_back(abs_ptr);
}
```

Listing 3.1: An example address-sensitive use of relative pointers

Due to the `std::sort` operation, perhaps for improved cache locality, the `rel_ptrs` will be iterated over in a different order in all the variants in the second **`for`** loop when absolute pointers are reconstructed from them. Therefore, the `printf` operation will immediately cause a divergence.

Passing a pointer value to a system call

One of the most straightforward ways of causing a program to exercise ASB is by using one of the many system calls that take a pointer value as one of their arguments. In current MVEEs, this will actually not be considered a divergence. According to Salamat's formalization of system call equivalence [19], pointer arguments are not checked for equality the same way this is done for non-pointer arguments, but rather the memory contents they refer to are compared. This specification inadvertently allows a bit of leeway for system calls to be considered equivalent even when their pointer arguments refer to different logical allocations, as long as the contents they refer to are equal. The snippet in Listing 3.2 demonstrates a program that currently does not cause a divergence in typical MVEEs, but would if stronger equivalence checks existed on pointer arguments.

```
const char string[] = "Always␣the␣same";
std::unordered_set<char*> strings;
for (int i = 0; i < 10 ; i++)
    strings.insert(strdup(string));
for (auto s : strings)
    write(STDOUT_FILENO, s, sizeof(string));
```

Listing 3.2: A snippet demonstrating the difference between pointer equivalence and allocation content equivalence checking.

Because `std::unordered_set` is most commonly implemented using a hash table, the iteration in the second **`for`** loop will happen in a different order, executing `write` system

calls that operate on different logical allocations. A current MVEE like GHUMVEE will not see any divergence though, as the allocations that are referred to have the same contents on every call. We found no prior research that investigates the security implications of this leniency. One class of attacks that could benefit from this are Time-Of-Check To Time-Of-Use (TOCTOU) attacks, where, in an MVEE context, the attacker tries to modify the contents of system call arguments in the small time window after they are checked by the monitor but before the system call completes [2]. Since this relaxed checking allows for any allocation to be validated by the monitor as long as its contents are equivalent to those in other variants, this broadens the range of logical allocations an attacker could potentially use to circumvent monitor checks. No research exploits currently exist that abuse this leniency though.
Note that the exception for pointer arguments in Salamat's formalization can already be regarded as a mitigation effort for ASB.

Passing a buffer containing a pointer value to a system call

When writing out a structure or buffer using a system call, a pointer to it is typically passed in as an argument. If that structure or buffer contained a pointer value, that is now present in the buffer passed to the system call and the monitor will detect that the buffers differ across variants. For some system calls, the layout of the buffer is completely unknown to the MVEE, such as for `write`: the character buffer that it receives could be obtained by taking the address of some **`struct`** containing a pointer as a field, but it could also be some string that is the same across variants. For other system calls, there is a documented layout for the buffer such as for `writev`, which takes an array of `iovec` **`struct`**s. This case is then reduced to the "passing a pointer to a system call" case, since the added difficulty of finding the pointer value in the buffer is absent. The pointer field of the `iovec` **`struct`** itself also refers to a buffer, with a similarly obscure layout to the one write takes. This highlights how one system call can incur multiple cases of ASB at the same time: in the `writev` case the pointer to the `iovec` **`struct`** buffer (1), the pointer contained in every `iovec` **`struct`** entry (2) and all the buffers that are referred to from within the `iovec` **`struct`** by its pointer field (3) can all differ across variants.

Formatting a pointer value into a string

A common debugging strategy is to print out pointer values formatted into strings, typically in hexadecimal format. In fact, when enabling the debug mode on the loader that GHUMVEE employs to achieve Disjoint Code Layouts (DCL) [10], the monitor immediately terminates when running multiple variants as the addresses where the stack is loaded in the variants are printed to the standard output. The reason why this happens is obvious: since the pointer values are different their string representations are different as well. This is another example of a whole secondary class of values possibly being containing variant-specific address data: every string could potentially contain a formatted pointer value, making every operation that manipulates strings a potential cause of divergences.
It's also worth pointing out the subtle way in which this conversion happens, characteristic of the *libc* functions providing this behaviour. They do not directly take a pointer argument to format it into a string but rather accept a variadic argument list that is then enumerated and interpreted based on a format string. This eliminates all type-safety

making it impossible to know which arguments were originally pointer values without call site context. Type checking variadic arguments has been the subject of research in the past [32], because they can be abused in a similar manner to a buffer overflow as part of an exploit.

Using an uninitialized variable

Automatic, register and heap-allocated variables in C11 are not default initialized [21, Section 6.7.9.10]. This means that, if not explicitly initialized, in a typical language implementation they will contain the memory contents of whatever part of the address space they occupy the moment they are read. For stack variables this depends on the depth and nature of the entire call stack up until that point, and for data allocated on the heap this is determined by what the contents were of whatever was previously allocated there, or typically 0 if the memory was freshly requested from the operating system[1]. Clearly, when the memory layout of a program is changed, there can be no guaranteed consistency across variants about which allocations occupy which previous allocations, or even which function frames occupy which memory region on the stack in the general case[2]. Therefore, these values could be different in the variants and using them in any way might yield different results every time, leading to divergences down the road.
In real world code bases the use of uninitialized variables is mostly accidental, although there exist cases where they are used as a low-entropy source of randomness, famously in OpenSSL, popularized by a faulty Debian patch from 2006 leading to a major security vulnerability until it was patched in 2008 [33].
In any case, every read from an uninitialized variable is undefined behaviour according to the C11 standard [21, Appendix J2] and various tools already exist that can detect [34, 35] or mitigate [36] it. This makes it stand out, as, contrary to most other ASB in this overview, it is already actively being investigated.

Passing a buffer containing any kind of padding to a system call

Very similarly to the pointer-in-buffer case, if the buffer being passed contains a structure that has been padded to satisfy the alignment of its members, that padding will typically be uninitialized and depend on the underlying memory layout to determine its value just like in the uninitialized variable case. This likely causes a divergence when passed to a system call, as the monitor will compare the contents of buffers as well.

3.2 Definition of ASB

We consider an operation address-sensitive if its output is in any way influenced by the memory layout of the program. Address-Sensitive Behaviour (ASB) is then defined as a sequence of address-sensitive operations. The use of *memory layout* instead of *pointer values* is important to also include the effects of uninitialized data in the definition. Because

[1] For security reasons, modern operating systems zero out pages when they are mapped by a process to avoid information leaks from another process.

[2] There have been MVEEs in the past that inserted different padding between function frames in all variants to protect against information leaks through uninitialized reads [5].

of this definition, all operations on pointer values are trivially considered ASB, as their effect is influenced by the value of the pointer. However, this influence does not necessarily cause a divergence. Consider the first few lines of the 64-bit `strlen` implementation of *glibc* [37] in Listing 3.3.

```
size_t strlen(const char * str) {
    const char *char_ptr;
    for (char_ptr = str;
            ((unsigned long)char_ptr & (7u)) != 0;
            ++char_ptr)
        if (*char_ptr == '\0')
            return char_ptr - str;
    // ... rest of implementation
}
```

Listing 3.3: First few lines of *glibc*'s 64-bit `strlen` implementation

This implementation aims to optimize `strlen` by comparing multiple bytes of the string at once to find a zero byte (not shown in Listing 3.3). This is only done starting from the first address in the string that is aligned on an 8-byte boundary, which is why this first loop checks for zero on a byte-granularity until that alignment is reached. That means that there exist 8 different control flow paths through this snippet for the same logical string in every variant, depending on how far the starting address of the string is from the next 8-byte boundary. However, from an outside perspective, this address-sensitive loop condition can never cause divergences since:

1. The returned string length will always be the same.
2. No global state is affected.
3. No system calls are made.
4. No invalid memory accesses happen[3].

Hence, address-sensitive operations such as the repeated alignment check on the string do not necessarily lead to divergences (i). Additionally, divergences are not always caused by ASB, trivially illustrated by the very exploits MVEEs are designed to protect against (ii). There do exist divergences caused by ASB though, as shown in the overview of real-world examples at the beginning of this chapter (iii). Therefore, the relationship between benign divergences and ASB can be modelled as in Figure 3.2.

[3]Provided that the string is null-terminated within the bounds of its allocation.

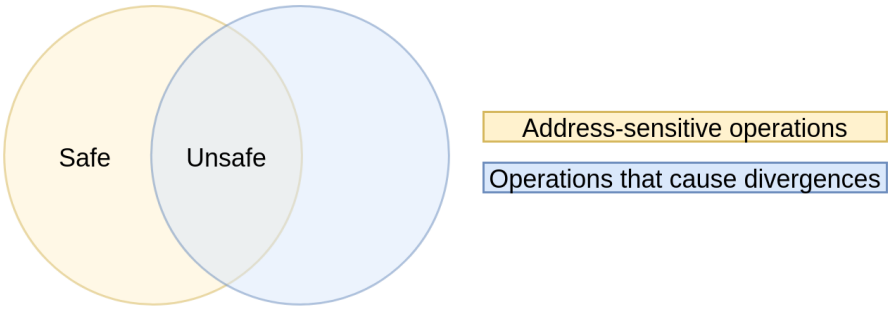


Figure 3.2: Relationship between the set of operations that cause ASB and those that cause divergences

Unsafe and *safe* ASB are defined in this context as operations that do and do not lead to benign divergences respectively. However, this is not always an inherent property of the operation itself. For example, if the loop body in the `strlen` example from Listing 3.3 contained a `printf` statement, that would be executed a different amount of times in the variants, causing a divergence. Yet, even if the `printf` statement is added, this `strlen` will never cause a divergence if it is only ever called for 8-byte aligned input strings. Hence, the safety of the alignment check is not solely determined by its isolated operation, but also by the code around it.

More generally, the only reason the `printf` statement was able to make this snippet unsafe *at all* is because the definition of a divergence is set the way it is, namely to system call equivalence. If full control-flow equivalence were required by the MVEE instead, the unmodified `strlen` could undoubtedly already cause divergences. A similar argument goes for MLD, as decreased diversification with more guarantees about consistent, non-API guaranteed alignment between the variants could change the safety of the loop.
This shows a fundamental trade-off between monitoring granularity, ASB and MLD: the coarser the monitoring granularity, the more ASB can be tolerated for a given amount of MLD and *vice versa*. One of the purposes of this work is therefore to evaluate what amount of MLD can be supported while retaining full ASB support in an MVEE with system call granular monitoring. At the end of Chapter 4, this matter is revisited in greater detail.

The context-dependence of the safety of ASB makes it hard to statically distinguish between safe and unsafe ASB in the general case. Consider the hash table example given earlier in this chapter: it is certain that it can exercise unsafe ASB in the two possible ways discussed, but it is not entirely clear at which point that ASB is actually introduced, or rather, precisely which operation is address-sensitive. One could argue it is the conversion from a pointer to an integer as part of the hashing computation. However, the resulting integer could be converted back to a pointer without any ASB in some other context. Other operations in the insertion process are subject to a similar argument about why they are not inherently address-sensitive: one could alternatively argue that the moment the index is computed and the key is inserted, ASB has occurred. Yet this could be the only key in the hash table, in which case there can be no address-dependent reallocation nor ordering. Additionally, the hash table could never be iterated over in the rest of the

program, or the reallocation condition could take only the absolute amount of inserted keys into account rather than their relative distribution over the buckets. Finally, even if the hash table contained multiple entries, really were iterated over and the reallocation condition was indeed address-sensitive, by sheer chance the index calculation could still put the logical allocations in the same order in all variants. Therefore, to statically determine whether this hash table is unsafe ASB, the memory layout of all variants would need to be predictable at compile time, counter-acting all MLD. In Chapter 4, we, therefore, focus on the effects of considering a piece of safe ASB as unsafe and *vice versa*.

From the examples presented at the beginning of this chapter it can be observed that two distinct types of variant-specific data are introduced into the program by MLD that cause benign divergences: address values and uninitialized data. Although their effect on the system call sequence can be similar, they differ greatly in both how they are used and why they are introduced into the program in the first place.
The purpose of MLD, as discussed in the background section, is to ensure that manipulations of memory at invalid locations, i.e. outside of logical allocations, yield different results in the variants. To that end, it diversifies the pointer values used in the variants such that invalid memory accesses cause divergences. However, this is only one semantic use of a pointer value, namely to refer to a location in memory. There exist many other semantic uses for pointer values or values derived therefrom in a program, such as to uniquely identify a certain allocation (hash table), signify an ordering in memory (relative pointers), or provide a pseudo-random value. These all inadvertently and inseparably get affected by the diversification as well, even though that was never the goal of MLD, causing benign divergences. Although it would be desirable to simply remove the diversification from an ASB perspective, this is not possible due to the intended effect it has from a security perspective.
There exists no such *intended effect* for the diversification of uninitialized data, which is an additional, unintended consequence of MLD. All of its uses that cause divergences are considered benign, hence "turning diversification off" for them can be done entirely without ramifications from a security perspective.
Consequently, both of these types of variant-specific data benefit from different insights. The analysis of ASB using pointer values has to consider both of their semantic uses and requires a close examination of the specific operations that pointers support. This is impossible to do while accounting for variant-specific values in their generality, as uninitialized data does not appear in these operations. Therefore, in the rest of this chapter, both are covered separately.

3.3 Analysis of uninitialized data

The address-sensitive use of uninitialized data includes both the examples about uninitialized variables and padding. Although they can be used in non-address sensitive ways, similarly to how alignment checks do not always lead to divergences, there is no harm in considering all of its uses to be unsafe[4], since the diversification they supply has no significant security benefit. In doing so, its very introduction into the program can equivalently be considered unsafe in itself. Therefore, this section will focus on the different ways in

[4] Apart perhaps from a performance perspective.

which uninitialized data can enter a program, rather than how it can be used to cause ASB.

All data is initially exposed to the program in an uninitialized except for variables with static and thread storage duration. The C11 standard requires them to be initialized to their stored value [21, Section 6.2.4] or essentially zeroed respectively prior to program startup and at thread creation [21, Section 6.7.9]. Since other code can run before program startup, these could technically be used in an uninitialized way but due to the implementation of the program load process adopted by most operating systems, this is not an issue in practice. In the Executable and Linkable Format (ELF), uninitialized static variables are placed in the `.bss` section of the executable which is zeroed out at load-time [38, p.15]. Initialized static variables are placed in the `.data` section and their initial value is encoded into the executable, so they are never uninitialized at all.
Other data such as the memory returned by calls to OS routines such as `mmap` [22] and `brk`/`sbrk` [39, p.64] are typically zero-initialized (except for file mappings) by default. Note that after a subsequent increase and decrease of the program break using `(s)brk` the returned memory's value *is* indeterminate [39, p.64].
The rest of this section overviews the methods in which uninitialized data can be acquired by the program.

Uninitialized memory obtained via a dynamic allocation routine

The sequence of allocated blocks that are returned by a dynamic allocation routine does not have a consistent ordering. Either it may be directly diversified by using a different allocator in the variants [5, 6] or it may be affected by other diversifications such as ASLR. This causes returned blocks of memory to occupy the space of different free blocks or regions of metadata with different previous memory contents. Or, according to the C11 standard:

> *The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions is unspecified. [...] The `malloc` function allocates space for an object whose size is specified by size and whose value is indeterminate.*

[21, Section 7.22.3]. The same goes for the other dynamic allocation routines, except `calloc`, which zero-initializes the memory.

Uninitialized stack data

Every time a new function frame is pushed onto the stack, the local variables contained in it are not automatically initialized [21, Section 6.7.9]. Rather, whatever memory was still there from the previous function call is typically still there. Even under only ASLR, *old* pointer values from a previous function frame can still influence some next frame's uninitialized variables. Additionally, a stack is an implementation detail that is nowhere defined by the C11 standard; segmented stack schemes, shadow stacks and stacks with reverse growth direction are examples of different implementations of this concept in different variants. Some of these have been used before to increase diversity [7, 8], further diversifying the content of uninitialized stack data. Therefore, no assumptions can be

made about the value of stack variables that are used before they are explicitly initialized. As mentioned before, doing so invokes undefined behaviour [21, Section 6.3.2.1.2].

3.4 Categorization of ASB caused by pointer values

Although it was determined earlier in this chapter that there can be no certitude about the safety of address-sensitive operations in the general case, there can be more detailed reasoning about safety for individual constructs. In the rest of this chapter, multiple categorizations are presented that aim to give insight in the ways ASB is introduced into a program and provide stronger guarantees about its safety in specific cases.
Note that the categorizations in this chapter only hold for ASB caused by pointer values, not by uninitialized data. For brevity, this qualification is omitted in this section and ASB is used to refer only to ASB caused by pointer values.

Categorization based on cause

It is clear that to have address-sensitive behaviour caused by pointer values, a necessary prerequisite is that at some point a value is accessible in the program that represents a memory address. Valid[5], diversified addresses can be obtained from 3 sources:

1. A dynamic allocation routine, both application-level and OS-level, such as `malloc`, `mmap`, or `brk`/`sbrk`. Routines that allocate on the stack such as the non-standard `alloca` [40] are also included here.
2. An address-of operation (`&`).
3. A function or array decaying into a pointer [21, Section 6.3.2.1].

As these values are of pointer type upon introduction[6], every possible form of ASB must start with that pointer value. There does not necessarily need to be any operation applied to it though. Passing a logical allocation containing a pointer value[7] as an argument to a system call is sufficient to cause a divergence. This could be either as a direct pointer argument, or as a struct or buffer containing a pointer value. For `writev` for example, it includes all the pointers and their buffers in the `iovec` **`struct`**s as well. The case where pointers are passed indirectly as part of buffer contents is revisited later in this chapter, when discussing the conversion of pointers to other types in general.

An example of an operation that is guaranteed to be safe is a dereference expression. It is interpreted as every expression whose evaluation translates to a memory access. This includes the star operator (`*`), the arrow operator (`->`) and the array indexing operator

[5] Other addresses, obtained by for example using a pointer without initializing it, are not considered as cause for ASB since they represent bugs in the program.

[6] Note that even though all memory addresses are of pointer type upon introduction, that does not mean that all values that are of pointer type in the program necessarily contain memory addresses. However, for analyzing the causes of ASB, this has no influence.

[7] Even though the address-of operator might be applied to the pointer value directly when the logical allocation *only* contains the pointer value, that case is a direct concretization of the more general "pointer in logical allocation" case. Hence, it is not considered separately.

(`[]`). The operation is technically address-sensitive since the memory it accesses directly depends on the address value used to refer to it. However, the only divergences that could occur from it are when the equivalent logical allocations do not contain the same contents in the variants or the used pointers do not point to the same logical allocations. None of these divergences are *caused* by this operation; they are all the result of either prior divergences or bugs in the program. Any subsequent divergence is attributed to them instead. Therefore, from an ASB perspective, the dereference expression can not introduce any benign divergences, so it must always be safe ASB. Note that a dereference expression is sometimes required or used to facilitate ASB in indirect ways, such as in conversions from pointers to other types (see later), but even there the result of this operation should be logically equivalent across variants. The source of ASB is then contributed to some other operation that appeared in the process instead.

Another series of operations are the arithmetic and comparison operations. They are well-defined as long as all pointers involved, including potential resulting pointers, refer to somewhere inside or one past the end of the same *array object* [21, Section 6.5.6.8]. When used within their well-defined form, these operations can never introduce ASB directly. A similar argument as for the dereference expression applies: if this operation does not involve memory addresses that refer to the same logical allocation in all variants, a prior divergence is blamed instead. However, contrary to the dereference expressions, when these operations are used outside of their well-defined behaviour using pointer values that refer to different logical allocations, they *can* be unsafe. A typical example of this is the use of relative pointers, of which an example was given in the overview at the beginning of this chapter. In itself, none of these operations necessarily influence the program to execute different system call sequences. However, since the results of these operations are of integral type (barring the addition of integral types to pointers), they increase the scope of potentially address-dependent values to integral types and all operations on them. The implications of this are considered in more detail later, when discussing the conversion of pointer values to integral types in its generality. Note that C++17 specifies the `std::less`-family [31, Section 23.14.7] of class templates, providing an *implementation-defined* specialization with a strict total order for pointer types. On a flat memory model, this is typically equivalent to comparing the numerical address values. In C, this is explicitly undefined behaviour, but typically yields the same effects.

Lastly, the single most common operation with which unsafe address-sensitivity can be introduced is by converting a pointer to some other type. Whenever that happens, those values become variant-specific as well and the scope of operations on variant-specific values increases. Although pointers can theoretically be converted to any type (not necessarily in a well-defined manner), a common conversion is to integer types. LLVM Intermediate Representation (IR) even has dedicated instructions to facilitate this in a cross-platform manner [41]. The problem with converting pointers to integers in the context of ASB is the broad use of integers for non-pointer-related purposes and the variety of arithmetic and other operations defined on them. Any ASB introduction analysis that allows these conversions to remain undetected must as such suddenly deal with a greatly expanded set of potential values in a program that may be tainted by address values, and a lot of them not even remotely having anything to do with the issue. However, detecting pointer to integer conversions is not as trivial as it seems due to the variety of ways in which it

can happen. Given that arbitrary arithmetic is defined on an integer, and integers can be compared to pointer values for equality, any search algorithm can be employed to search the space of integer values and find one that compares equal to the pointer value. This suggests that the real scope of possibilities for expressing pointer to integer conversions is theoretically infinite. Listing 3.4 gives an illustrative example.

```
char* some_ptr = "";
size_t intptr = 0;
while (((char*)intptr) < some_ptr)
    intptr++;
```

Listing 3.4: An esoteric way of converting a pointer to an integer.

This is an example of a linear search algorithm through integer space, incrementally trying different integer values until one is found that compares equal to some pointer value. Although this will definitely compile and give the correct result for common language implementations on flat-memory architectures, it is rather contrived. In a real world code base, one of five techniques is typically employed. An overview of these is given here.

1. Casting. The most straightforward way to convert pointers to integers is through the use of type-casting. Although not the same as a cast, implicit contextual conversions to integers such as for example assigning a pointer to an integer are included in this category as well. This is generally implementation-defined behaviour in C11 [21, Section 6.3.2.3] and C++17 [31, Section 6.7.4.3], except for those integer types that are specifically designed to be large enough to hold pointer values on every platform (`intptr_t` and `uintptr_t`) [21, Section 7.20.1.4]. Even then, there exists no portable guarantee for the actual contents of that integer after such a conversion, apart from ensuring that if the integer is converted back to the same pointer type without any operation applied on it, the resulting pointer value will not differ from its initial value. Nevertheless, casting pointers to integers and back is a common practice [42] and on a flat memory model most implementations guarantee that the integer representation is bit-wise equal to its pointer counterpart [41]. This is relied upon for example when checking for alignment as discussed in the beginning of this chapter.

2. Using pointer arithmetic. As previously discussed, the subtraction of two pointer values is only well-defined if both point into the same *array object* [31, Section 8.7.5] [21, Section 6.5.6.8]. However, the result will typically simply be the difference of their numerical address values. Therefore, the result of subtracting the `NULL` pointer from another pointer is equivalent to the integer representation of that other value. Similarly, an integer can be converted back to a pointer by adding it to the `NULL` pointer. In *Gnulib*'s `obstacks` implementation [43], the `__PTR_ALIGN` macro uses this trick.

3. By reinterpreting a pointer to the memory this pointer resides in. The address of any logical allocation can be taken with the `&` operator, cast to a pointer to any type and subsequently dereferenced. This violates the strict aliasing rule in C11 [21, Section 6.5.7] and C++17 [31, Section 6.10.8] but just reinterprets the memory content in

most language implementations. In fact, when a pointer is being reinterpreted to an integer in this way, Clang will generate the same IR for simple cases. This can be done both via dereference + assignment and via `memcpy`.
The example about passing a pointer to a structure that contains a pointer as one of its fields to the `write` system call is also a use case of this: the address of the allocation in which the pointer resides is taken, cast to a pointer to some other type, typically **`char`**, and the pointer is reinterpreted as an array of **`char`**s together with the other fields. This yields an array that contains address-dependent data which, without context, is indistinguishable from a regular array that is the same across variants.

4. Through a union. A union is a data structure whose members all occupy the same space in memory [21, Section 6.2.5.20]. Therefore, when a union contains both an integer and a pointer and the pointer member is active, any read from the integer will typically result in an integer representation of the address value:

   > *If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called "type punning"). This might be a trap representation.*

   [21, Section 6.5.2.3.3]. Conversely, in C++17 this same operation is undefined behaviour [31, Section 12.3], but behaves no differently in practice than C11.

5. Using variadic arguments. As discussed in the example where a pointer is converted to a string, passing a pointer to a function that takes a variadic argument list removes all type-safety and the types of the variadic arguments must be reconstructed inside the function through external means. The most common case is the `printf`-family of functions, where a format string is used to reconstruct the argument types.
   Hence, by using `va_arg` with integer type on an argument that originally had a pointer type, a conversion can be achieved, and *vice versa*.

Categorization of ASB based on type of address-dependence

By simply defining any monitoring granularity, it may be guaranteed that ASB is already introduced in the program. In the case of system call granularity, most system calls cannot be meaningfully executed introducing ASB simply because they take pointer values as arguments. There is no way to not have ASB in this case, as it happens to be API-specified that address-dependent values must exist among the regions of program state that are inspected by the monitor to determine equivalence. The pointers are just being passed as arguments to a function though, in essence not dissimilar from any other function[8] except that other function calls are not explicitly monitored.

Concretely, this applies to pointer arguments to system calls, both as direct arguments and as API-defined members of struct arguments. Calling any system call with these types of arguments is categorized as *weak ASB* because it shows the weakest dependence

[8] Disregarding the differences between system calls and ordinary function calls, which are not relevant for this rationale.

on the numerical value of the pointer: it is simply used to refer to some logical allocation. Its value is otherwise never specially inspected by the program. Weak ASB can only occur directly at the system call invocation by definition. As discussed in the pointer-as-argument example at the beginning of this chapter, current MVEEs that use Salamat's formalization of system call equivalence are insensitive to this, as they do not consider the pointer values themselves to determine equivalence, but the memory contents they refer to.

Note that pointer values contained in a buffer unknown to the system call are not considered weak ASB. This is because the system call API does not specify that that buffer contains a pointer value at all, so it is the *use* of the system call that is memory-layout dependent, not its very definition. Although this difference in use seemingly also exists for explicit pointer arguments that may have the `NULL` value which, if consistent across variants, makes the system call non-address-sensitive, this is a documented API feature that is explicitly supported by the system call and will correspond to different functionality, e.g. `NULL` as first argument to `mmap`. This is not the case for buffer arguments: the pointer values potentially contained within them are completely unknown to the system call and it will not provide different functionality based on their existence.

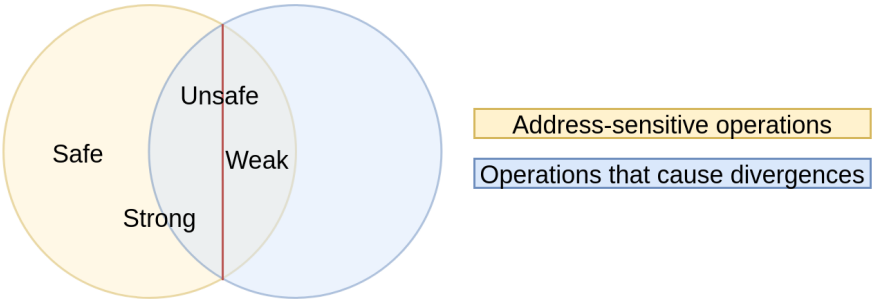


Figure 3.3: Relationship between strong/weak ASB and operations that cause divergences

Conversely, some divergences seem to signify a more profound address-dependence than weak ASB and use the pointer for its numerical address value rather than to merely refer to some logical allocation as a function argument. In the examples at the beginning of Chapter 3, the iteration over a hash map causes completely different system calls to be executed in the variants, or the same system calls to have non-equivalent pointer and non-pointers arguments. This is because the iteration happens in a different order in all the variants, fundamentally due to the address-dependent insertion method. This operation solely uses the pointer for the numerical value it holds and does not consider the logical allocation that the address represents. We call this a *strong* dependence of the program on the specific address that a pointer variable holds. It shows itself in divergences by influencing non-pointer arguments and buffer contents and changes the logical allocations both explicit and implicit pointer arguments refer to. Pointers that are implicitly passed as buffer contents are also included in this because their presence in a buffer hides their pointer nature, influencing program state beyond pointer values alone which can never happen in weak ASB.
Current MVEEs can only recognize and support some pre-defined diverging system call patterns caused by strong ASB, such as those in `ptmalloc` [2]: the program depends on

some numerical property of the address value, namely that it is divisible by 1MB.

Note that weak address-sensitivity is a subsidiary effect, with strong address dependence taking precedence in case both are present. For example, if the monitor detects that a system call operates on different logical allocations in the variants[9] it is not considered weak ASB, even though the system call takes pointer arguments, which is the definition of weak ASB. The strong ASB that led to this non-equivalent pointer argument takes precedence in the categorization of this divergence.

Since weak ASB never leads to divergences under Salamat's formalization of system call equivalence, it is guaranteed safe ASB. For strong ASB, such guarantees do not exist; as discussed previously, it cannot be determined from the use of, for example, a hash table whether this will lead to a divergence or not. The relation between ASB and divergences from Figure 3.2 is further extended using this insight in Figure 3.3.
Additionally, the differences between strong and weak ASB are summarized in Table 3.1.

Weak address dependence	**Strong address dependence**
Pointer values are used for their access to a logical allocation	Pointer values are used for their specific memory address value
Introduced only at system call invocations	Introduced anywhere in the program
The different values are all of pointer type	The diverging values can be of any type (including pointer type)
Never causes a divergence, safe by definition	Does not necessarily introduce divergences
Mitigated by current MVEEs	Not supported[10] by current MVEEs

Table 3.1: Overview of differences between strong and weak ASB.

Within strong ASB, the details of its dependence on address values can still further be qualified. In the rest of this section, a further categorization is presented that analyzes this dependence based on how functionally resilient the program is to removing it.

In the hash map example, the index computed from a pointer value is merely used to determine a bucket to place the pointer in, and subsequently disappears from the program. At no point is it used to reconstruct the original pointer value. This means that as soon as an address value is not a dereferenceable entity in the program anymore, it can technically have any value at all as long as the same hash value is always returned for the same pointer operand. Hence, all pointers could hash to the same value without compromising the functionality of the hash table[11]. The combination of assumptions that the

[9] Assuming the divergence is caused by ASB.

[10] Barring hard-coded exceptions such as `ptmalloc` in GHUMVEE [2].

[11] The performance of the hash table will suffer greatly if all pointers hash to the same value, because of the amount of collisions. Performance effects or time-complexity guarantees about the program are not considered part of its functionality though. Nonetheless, an alternative hash operation that preserves these guarantees but still proves that there need not be any numerical dependence on the pointer value would be to generate and cache a random number for every pointer insertion. The same random number would

program makes about the numerical value of a pointer based on a non-address value that was derived from it (index) is referred to as the *feedback* of the derived value on the pointer value. Hence, in the hash table case, this feedback is non-existent, since the index is only used to determine the position of the pointer in the table, and makes no assumptions about any value for the pointer itself. It is a strictly one-directional operation. If a case of strong ASB incurs no feedback, it is called *non-reversed strong ASB*: the program does not depend on any numerical relation between the hashed/index value and the pointer value; there is no later decoding of the derived value that is supposed to satisfy some assumption.

This is not always the case though: in the relative pointers example from Listing 3.1, the derived value is the difference between two pointers which is later added back to one of the pointers to reconstruct the other pointer. This implies maximum feedback from the derived value back to the reconstructed pointer value: the program assumes that the sum of the derived value and the pointer value exactly matches the value of the other pointer. In fact, there exists only one value for the derived value that will not violate the program's assumptions, which is the actual variant-specific difference between the pointer values. This is the complete opposite scenario of the hash table case, where no derived value could be constructed that would compromise the program's functionality, as the program made no assumptions that could be violated in the first place.

Both of these cases represent the extremes between which feedback can exist from derived values to pointer values. In other cases, it can be more nuanced. Consider the memory alignment case: the derived value is essentially some number qualifying whether or not this pointer is aligned to a certain boundary. This could be an integer containing the difference to a neighbouring multiple of that boundary, or a simple boolean. A typical use case is to then use that derived value to modify the pointer value again until it is aligned, and then do something with the memory it refers to. The `strlen` example from Listing 3.3 shows this very well: the pointer is incremented only if the derived value indicates that the pointer value is not aligned (feedback), and the memory is accessed on every iteration.

In this specific case the alignment on `char_ptr` is performed because some vector instructions used for processing 8 bytes of the string at once require that the memory on which they operate is suitably aligned [44] to an 8-byte boundary. To evaluate the influence that the result of the alignment check, i.e., the address-sensitive operation, has on the correctness of the program, consider the situation where we provide a *bogus* result, chosen by us, for the alignment check ourselves. By examining the effect that has on the program, we can quantify the amount of feedback that exists from the alignment check to the pointer value.

First of all, whenever the bogus alignment check says the pointer is aligned to an 8-byte boundary while it is not, we violate the programs assumption later on that the instructions that require this alignment do not crash. And if the vector instructions in this routine were implemented using inline assembly or compiler intrinsics, a crash would definitely occur. However, likely for portability of this code between systems with different word sizes[12]

be returned if the same pointer was hashed again.

[12]The snippet in Listing 3.3 was modified to assume 64-bit for simplicity, the original also supports 32-bit.

and CPU capabilities, this is not the case, i.e. no inline assembly or intrinsic functions are used. The **`char`** array is interpreted as an array of aligned 64-bit integers instead, which are checked for zero-bytes as a whole, i.e. 8 bytes at a time, using a clever magic number trick[13]. The idea is here to provide a Single Instruction Multiple Data (SIMD) algorithm that the compiler's optimizer can easily vectorize if the target platform supports it. If the compiler cannot prove that `char_ptr` will be aligned to 8 bytes after the loop, it will use unaligned vector instructions, or none at all. Hence, if a bogus value is provided at compile time, the compiler will not trust that `char_ptr` is suitably aligned anymore and the unaligned `char_ptr` will not cause any issues.

Except, this is not the only reason why `char_ptr` needs to be aligned. As the string is now checked 8 bytes at a time but the zero-terminating byte is not necessarily in the last byte of those 8 bytes, the last check could read at most 7 bytes past the end of the string. This will not cause a segmentation fault as long as all of those 7 bytes are still contained within a mapped page. This is only guaranteed when every 8-byte access in the string is aligned on an 8-byte boundary[14]. Since the string is iterated 8-bytes at a time, this constrains `char_ptr` to be aligned to 8-bytes at the beginning of the iteration, i.e. after the initial alignment loop. Hence, in case of the the bogus alignment check, there exists a chance that the last 8-byte access will fault depending on the length and location of the string, crashing the program.

However, the above consequences are not guaranteed to happen for every possible combination of bogus values. If the bogus value over-states the actual alignment of `char_ptr`, the above scenario unfolds. However, if the bogus value consistently denies that `char_ptr` is suitably aligned, the initial alignment loop will never be exited and the `strlen` will be calculated using a byte-granular search, without error. In fact, as long as the bogus value is never true when the string is not actually aligned, the `strlen` will work.
This places the amount of feedback from the alignment result back to the pointer value somewhere between the hash table and the relative pointers: if the pointer is aligned, the bogus value could be anything but if the pointer is not aligned, the bogus value must not over-state its alignment.

This analysis shows how the amount of feedback of an address-sensitive operation is highly context-dependent and not necessarily all-or-nothing as in the first two examples. If there exists any feedback at all, this is called *reversed strong ASB*.

Note that the qualification of feedback and whether or not some ASB is reversed depends on the specific granularity at which pointer operations are observed. In the relative pointers example from Listing 3.1 the sort operation is hidden behind a library call, naturally causing an observer to focus on the effects of changing the values that are stored in the collection of relative pointers, and evaluating their feedback on the reconstructed pointer values. However, if the relative pointers were stored in their variant-specific form and the sort operation was investigated in detail, a similar situation to the hash table appears:

[13] The actual logic behind how a single zero byte can be detected using vector instructions is out of scope for this work.

[14] And the page size must be a multiple of 8.

the relative pointer could be swapped out for a bogus value right when it is compared to another value or involved in some other operation that determines its position in the collection. If this is consistently enforced throughout the program, it would be transparent to the programmer that the relative pointers are not in fact stored in numerical order. Hence, there would be no feedback to the rest of the program.

It is unclear whether reversed strong ASB can always be reduced to a case of non-reversed strong ASB at a finer granularity in this way. Even in this simple case, it is not straightforward to detect at which granularity the bogus value should be provided, and to foresee which additional precautions should be taken to ensure that the programmer's expectation of the numerical order of the elements in `rel_ptrs` is preserved. Technically, the *non-reversed* qualification of ASB in the hash table case is only based on the assumption that the hash operation is opaque to the programmer in the first place. If the numerical pointer value is simply considered its own hash value and the programmer is aware of this, the program might make similar assumptions to the relative pointers case about the order of the pointer values in the hash table. Likewise, the seemingly non-reversed pointer value output of `printf` to `stdout` could technically also be read again, potentially by a human observer, and parsed back into a pointer value. This strongly suggest that all ASB can be made into reversed ASB, but not all reversed ASB can be reliably made into non-reversed ASB. The terms *feedback* and *reversed ASB* are still useful though to describe the influence that a value change of a derived pointer value has on the functionality of the program.

Categorization of ASB based on control flow effect

Sometimes, an instance of ASB taints the control flow of a program. This means that at one point, something causes at least one variant follow a different control-flow path. A necessary prerequisite for this is the diverging evaluation of a certain condition, which, in case this divergence is caused by ASB, must somehow have been *tainted* by an address value. Note that such a tainted conditional evaluation is not necessarily a sufficient prerequisite for a divergence, as illustrated by the `strlen` snippet in Listing 3.3. This control-flow altering ASB clearly requires strong ASB, as a pointer has to be interpreted for its numerical value to taint the evaluation of a condition. Additionally, weak ASB only occurs at system call boundaries, which are not conditional branches. Therefore, weak ASB can never lead to control-flow altering ASB, but strong ASB *can* exist outside of control-flow altering ASB. This is demonstrated by a rather contrived $O(n)$ implementation of the `sort` operation in the relative pointers example from Listing 3.1, demonstrated in Listing 3.5[15].

[15]This implementation technically contains a memory error where, if `data` does not contain `UINT32_MAX`, there will be written one-past the end of the array. One could say it is an API requirement that `data` should have an extra element allocated, but to prove that the memory error is not inherent to the claim that strong ASB does not necessarily alter control flow, an error-free version is attached in Appendix A. We did not include it here to avoid obscuring the point of the snippet.

```
void sort(uint32_t* data, size_t len) {
    static std::vector<bool> cnts(UINT32_MAX);
    std::fill(cnts.begin(), cnts.end(), false);
    for (size_t i = 0; i < len; i++)
        cnts[data[i]] = true;
    for (size_t i = 0, j = 0; i < cnts.size(); i++) {
        data[j] = i;
        j += cnts[i];
    }
}
```

Listing 3.5: A counting sort for 32-bit numbers with data-independent control flow

This specific implementation assumes that `data` has distinct elements, which is realistic for the use case of relative pointers. The only conditional evaluations are of `len` and `cnts.size()`, both of which are variant-agnostic. Its time-complexity is linear, but a big constant term caused by the second loop limits its usefulness in practice.
A less contrived example would be a call to `printf` with a pointer argument, which will also not cause diverging control flow in the variants although it is strong ASB.
Note that the notion of *control flow* is interpreted broadly here, not only referring to conditional jump instructions that might send variants down different branches but also including conditional move instructions [45, p.50] as they are expressed similarly in C/C++ source code, which is the target of this work.

3.5 Visual representation

Finally, to conclude the categorization of ASB and this chapter, all of the introduced terminology and concepts are related visually in Figure 3.4.

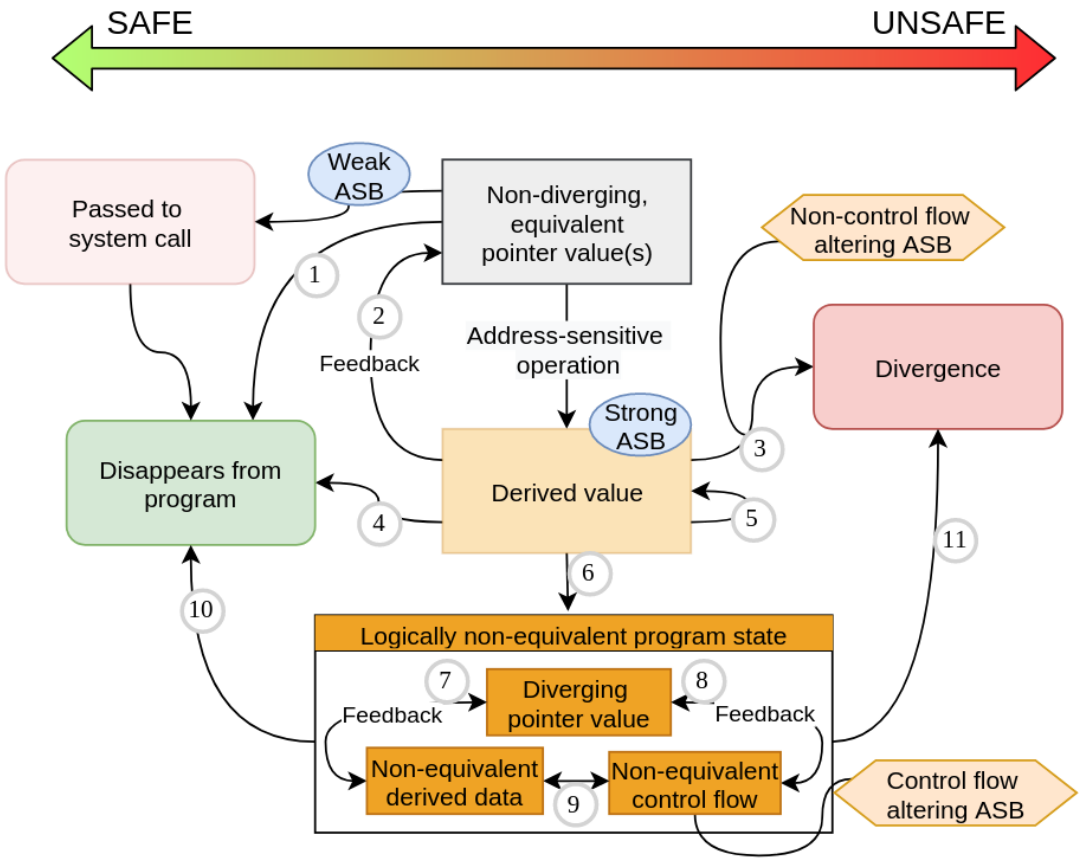


Figure 3.4: Flow of diversified address data through a program from an ASB perspective.

The term *non-diverging, equivalent pointer value* is introduced to disambiguate between pointer values that refer to different logical bytes in the variants, and pointers that do not. If the same logical pointer variable refers to the same logical allocation in all variants, this is a non-diverging pointer value that is equivalent in the variants. Every address value starts out as such. If it refers to a different logical allocation in some variants, or a different offset into the same logical allocation, we call it a *diverging pointer value*, to highlight how it is a first step towards an eventual divergence, since subsequent memory accesses through that pointer variable will have different effects in the variants.
The arrows between the various states in Figure 3.4 are explained below:

1. This occurs when a non-diverging, non-escaped pointer value simply disappears from the program by leaving the scope.

2. A derived value is the difference between two pointers for example. By reconstructing an absolute pointer from it, a non-diverging pointer value is obtained.

3. A derived value is passed to a system call. The most notable example of this is the pointer-in-buffer or pointer-in-string case, where a pointer value is transformed into a derived value, i.e. **`char`** buffer or string representation, before being passed into a system call.

4. Similarly to 1, a derived value can simply disappear from the program if nothing

references it any longer.

5. A derived value can be converted into another derived value. For example, a relative pointer undergoes pointer arithmetic. After reconstruction, it will be as if the arithmetic was applied on the absolute pointer instead.

6. If the derived value is the distance of a pointer to a next alignment boundary, such as in the `ptmalloc` example from Figure 3.1, it can be added to a non-diverging pointer value to produce a logically diverging pointer value that is aligned to that boundary in all variants. Alternatively, the result of an alignment check could be different in the variants, leading to different control flow.

7. A diverging pointer value can produce a derived value similarly to a non-diverging pointer value. This could be a relative pointer, alignment result, etc.

8. An alignment check on a diverging pointer value will lead to non-equivalent control flow.

9. Similar to 8.

10. Similar to 4 and 1. A program might go into a logically non-equivalent state without ever resulting in a noticeable divergence from the monitor's perspective. For example, a list of pointers could be sorted in ascending order without ever accessing them. Or the example from Listing 3.2, where the hash table of pointers is iterated over in variant-specific order but, because the allocation contents are the same, no divergence is observed by the monitor.

11. A diverging pointer value or piece of non-equivalent derived data is passed to a system call, or non-equivalent control flow in the variants leads to a divergence. Iteration over a hash table is an example of this.

Chapter 4

Overview of mitigation approaches

In general, there can be two perspectives to the problem of eliminating the benign divergences caused by ASB: determining whether a divergence is a false-positive from within the monitor, or preventing false-positive divergences in the first place.
From a monitor perspective, it is hard to determine whether a diverging system call is benign or not. The only possibility would be trace back the specific diverging system call arguments to the root of their divergence and comparing their provenance. This is only possible if dynamic taint analysis is used to track all pointer values during execution of every variant in the MVEE. Additionally, it should be provable for the monitor that diverging values are *only* tainted by false-positive sources of non-determinism like pointer values. This would also require tracking all untrusted program input, which, even disregarding its performance overhead and false positive/negative rate, would be an entirely different security scheme to the one an MVEE offers. Therefore, this angle is not further entertained as a mitigation approach.
At the point where arguments to system calls diverge, this work considers any divergence to be indistinguishable from a malicious divergence in the general case. Hence, the challenge shifts towards preventing false-positive divergences in the first place, without preventing true-positive ones at the same time. In fact, the following metrics are considered when evaluating an approach, loosely in decreasing order of priority:

1. **Completeness**. As a first all-round look into ASB, the provided solution should be able to cover as much real-life ASB as possible.

2. **Soundness**. The solution should not hide any true positive divergences and it should introduce little to no new false-positives or bugs into the variants. This is especially relevant in the context of undefined behaviour: a strict interpretation of the language standard would allow the mitigation solution to behave arbitrarily but ideally, the *de facto* behaviour that the programmer expects should be preserved. On the other hand, as long as the program's functionality or security is not altered, there is no inherent problem with a lack of soundness.

3. **Manual developer effort required**. From an MVEE perspective, the more effort has to be put into manually preparing code before it can run in the MVEE, the smaller chance of adoption the system has. Secondly, the less manual effort required for a solution, the more insight into the ASB problem it demonstrates, indicating a more interesting angle to pursue in future work.

4. **Flexibility**. An ideal solution is agnostic to the granularity of the applied memory layout diversification. In practice, this means supporting the randomization of the base address of every logical allocation.

5. **Security**. This work considers ASB mitigation in the context of MVEEs, so any covert channels [2] the solution opens across variants that reverse/neutralize the applied diversification are undesirable.

6. **Run-time and memory overhead**. Modern MVEEs can be very efficient, depending on the supported parallelism of the underlying hardware and the system call intensity of the target program [1]. Any degradation of this metric decreases the likelihood of adoption, but since this work presents an initial step towards ASB mitigation this is not the main focus.

Note that there exists some overlap between the metrics presented here: a lack of soundness may for example increase performance overhead, decrease security and introduce bugs into the program. These three specifically are referred to as *secondary metrics* below.

As discussed in the previous chapter and contrary to ASB caused by address values, mitigating ASB caused by uninitialized data does not have to consider the intention of the MLD at all. Therefore, it is covered separately, after which the bigger problem of mitigating address-sensitivity caused by the use of pointer values is considered. All of the work here assumes that full source code for the entire program is available, including all dependencies.

4.1 Mitigating ASB caused by the use of uninitialized data

The analysis provided Chapter 3 argues that there is no harm in considering all uses of uninitialized data to be ASB and, therefore, considering its very introduction into a program to be ASB. The straightforward way of mitigation would then be to make sure all uninitialized data has the same value across variants, e.g. zero [46]. However, this solution over-extends its purpose: the value of uninitialized data should merely be the same in the variants, not necessarily have a predictable value. Some uses of uninitialized data are not accidental, but intended as a low-entropy source of randomness, such as in the OpenSSL example given in Chapter 3. Since this is not considered a secure or even reliable source of randomness and already actively discouraged [47], it is reasonable to assume it is either combined with other sources of randomness to produce the eventual result, as in the OpenSSL example, or not used at all.

Another example is the `__gen_tempname` function from *glibc* in `tempname.c`, where a value is used without initialization as part of the process to generate a random filename. The case where the value is 0 is explicitly considered as an edge case though, upon which it is initialized with a random value.

Both of these examples seem to suggest that real-world programs do not functionally depend on reading a different value from uninitialized memory, and account for its inherent unreliability. Zeroing out uninitialized data thus seems like a realistic solution, and there is no need to over-complicate a mitigation solution by attempting to provide a random value to initialize the data with.

This solves all the uninitialized data examples given in the overview at the beginning of Chapter 3; if all sources of uninitialized data covered in Section 3.3 are zeroed after allocation they can never cause benign divergences. The implementation of this instrumentation is covered in Chapter 5.

4.2 Mitigating ASB caused by the use of pointer values

As highlighted by its categorization, ASB caused by pointer values is introduced at some point in the program and may not have an effect until an arbitrarily distant and generally unpredictable future point. Additionally, a source of ASB such as an alignment check can not only lead to a single divergence down the line. Wildly different code may be executed when the control flow of two variants is allowed to differ, potentially causing different modifications to program state to cause a multitude of divergences down the line. Volckaert has previously referred to this as a *ripple effect* in the variants [2]. For these reasons, an ideal solution focuses on detecting the earliest point at which ASB is introduced and makes sure its effect across variants is the same there, before it can cascade and multiply into other parts of the program state, where it becomes uncontrollable [2].

High-level strategy

The easiest and least invasive way to mitigate an address-sensitive operation is by providing the same input to it in all variants. This entails the design of a mapping from variant-specific values to variant-agnostic or *unified* values, that is used to instrument the input to address-sensitive operations.
Secondly, some address-sensitive operations must be selected to be instrumented. Perhaps the most obvious solution would be to use taint-tracking to find out which operations are unsafe. However the taint tracking necessary for this kind of analysis is of the most complex kind. Both explicit and implicit flow [48] need to be captured, e.g. during an alignment check, both the explicit flow from the pointer value to the boolean indicating its alignment status must be covered, as well as the implicit flow resulting from both branches of the check. The analysis additionally needs to be inter-procedural [49] and needs to completely and soundly resolve all alias analysis [50, 51] to be useful in practical applications. Consider the common case of ASB; the hash table case: the point where pointers are stored in the table is arbitrarily far from the point where iteration takes place. The crucial information that makes the iteration address-sensitive is in the insertion procedure though, meaning that an analysis would need to carry the information that the order in which the pointers are stored is address sensitive throughout the whole program. This set of requirements can not be met by any static analysis tool due to its inherent undecidability [52].

For the problem of mitigating ASB however, a sound analysis is not necessarily required: as long as all ASB is mitigated, it is sufficient that the additional mitigation performed does not lead to bugs. The issue of instrumenting the target program without breaking it thus persists whether a static analysis tool or a *ball-park* over-estimating assumption about which operations are unsafe is used. This work will opt for the latter, whose soundness can optionally be enhanced by static analysis techniques, without relying on their

precision for completeness.

4.2.1 Choosing an over-estimation of address-sensitive operations

As previously discussed, any automatic instrumentation of specific pointer operations must deal with the side-effects it may have given that the program still considers the *unified* value to be derived from a valid pointer value. What those side-effects are depends on which operations are instrumented. The selection of operations that are considered unsafe and the subsequent instrumentation effectively introduce an *ASB boundary* into the program between the variant-specific form of memory addresses and their unified form. After that boundary is crossed, ASB can no longer occur. This section evaluates the various considerations that go into drawing that boundary, and its effects on program correctness, security, performance and ASB coverage. The existence of a unified address space is assumed for the discussion below, to and from which variant-specific addresses can be freely translated.

Pointer-granular detection: every use of a pointer value is unsafe

Given that Chapter 3 suggested that there are many ways, including esoteric ones, to convert a pointer value to an integer, the only ASB boundary that is guaranteed to be complete considers the very introduction of pointer values into the program to be ASB. All sources of pointer values in a program, listed in section 3.4, can be wrapped to provide the same unified value in all variants for each logical allocation. This trivially eliminates all ASB from the program, because all pointers will have the same value in all variants. However, these unified addresses will not necessarily be valid memory addresses and any use of them to read from or write to the memory they are supposed to reference will be invalid in all variants[1]. Hence, on every memory access that unified address must be specially translated back, i.e. *deunified*, into the variant-specific address that was originally obtained for that logical allocation. This detection strategy is said to have *pointer granularity*, as it considers every use of a pointer unsafe.
There are a few downsides to this approach:

1. Run-time overhead: load/store operations are the most commonly executed instructions [53]. Apart from non-array stack variables whose address is never taken, every operation on memory will have to be wrapped with a deunification operation, slowing down the program considerably.

2. Security: because pointers are now always the same in the variants, some memory error exploitations that the MVEE is designed to catch might start to show similar effects in all of them, limiting the MVEE's ability to catch their exploitation. Out-of-bounds reads or writes using pointer arithmetic and subsequent dereference expressions are supposed to access completely different logical memory locations in the variants, hindering their exploitation. However, since the memory layout of the unified address space is the same across all variants, these invalid accesses may

[1]Note that this is only a problem from a program correctness perspective. The fact that a unified value could be a valid memory address in all variants does not decrease security as the memory layout is diversified, so the access will have different consequences in all variants.

actually access the same logical allocations when this mitigation is applied. Consequently, they will have the same effects in the variants, counteracting the intended effect of MLD and essentially *hiding* a true positive divergence.

3. Introduction of bugs: in the `strlen` example from Listing 3.3, this mitigation strategy could cause both bugs discussed in Section 3.4, as the alignment check might return bogus values relative to the actual alignment of the deunified value, depending on how the bogus values are chosen.

Note that this approach is not necessarily equivalent to disabling all MLD in the first place and just forcing all logical allocations to the same positions in all variants. Consider the classic stack smashing attack, where the return value of a function frame is overwritten such that on a `ret` instruction, the program jumps to some attacker-controlled position in the code [54]. The memory access that a `ret` instruction incurs is never explicitly written in source code. The load/store instrumentation thus never has to be applied to it and its value will always have to be variant-specific. Without any MLD, this is still exploitable but in an MVEE with full MLD including DCL this approach does not decrease security in this specific case.

Operation-granular detection: specific uses of pointer values are unsafe

A distinction was made in Section 3.4 between safe and unsafe ASB. Therefore, the analysis could be refined by only considering unsafe operations, rather than all of them. In this new approach with *operation granularity*, pointer values are allowed to exist in a variant-specific way until they are used in one of these operations and replaced by their unified value. Since the dereference expression is not one of such unsafe operations, the load/store overhead is avoided. In the other cases, the following instrumentation rules are applied:

1. For pointer comparisons: the comparison is done on their unified value instead. For equality comparisons and comparisons within the same logical allocation, this is this equivalent to the non-instrumented version.

2. For pointer subtractions: the difference between their unified values is returned instead. This will always have the same sign as the programmer expects due to rule 1. For pointer subtractions within the same logical allocation, this is equivalent to the non-instrumented version.

3. For integers added to pointers: the integer is added to the unified pointer value instead, then the deunified value of the result is returned. If the result does not exceed allocation boundaries, this is equivalent to the non-instrumented version. If it does, it is assumed the integer was obtained using rule 2, meaning that it is a relative pointer and its addition to the pointer value is a reconstruction of an absolute pointer. After deunification, the result will thus always represent the same logical allocation in all variants.

4. For conversions of pointers to integer: the unified address value is returned instead. To avoid bugs, conversions from integers back to pointers apply a deunification procedure.

Since (1) the only additional operation on pointers outside of the aforementioned, namely a dereference operation, can not be unsafe, (2) these operations lie at the basis of all unsafe ASB and (3) the returned value of each of these operations is the same across variants, no unsafe ASB can occur if these four rules are enforced (1, 2, 3). Although detecting rule 1 through 3 can be done reliably as they are always explicitly expressed in source code in the same way, the broad scope of conversion possibilities between pointers and integers suggested by Section 3.4 provides a compelling argument that rule 4 can theoretically not be detected as precisely in the general case. This might not impede this approach as much in practice though, if the most common ways of converting pointers to integers previously highlighted in this work are detected.

Pointers are not only converted to integers however, but also to other types like **char** arrays. The canonical divergence because of this was described in the overview of examples at the beginning of Chapter 3, namely writing out buffers containing pointer values. As the conversion typically does not happen directly on the pointer value, but rather on some logical allocation that contains it, reliably detecting when this happens and unifying the pointer value accordingly is not always straightforward. Consider the case where a pointer to a struct is converted to a **void***: this might be because it is stored in some type-agnostic collection and cast back to the correct struct pointer before every use, or it might be to later cast to a **char*** and written out using `write`. Simply detecting every type conversion from pointers to pointers and pointers to structs is both complex and performance-unfriendly: the type that is cast to may point to a struct with similar layout, upon which no unification should be performed, and most conversions from **struct*** to **void*** and back are just to use the struct in a type-agnostic way, what with **void*** being the idiomatic type to reference any type. Applying an unnecessary unification operation every time would be wasteful.

Outside of the divergence caused by the pointer-in-buffer case, there is limited applicability for other imaginable unsafe uses of this conversion construct in practice. Therefore, the assumption could be made that unsafe ASB caused by reinterpreting a pointer to be a **char** array always manifests through diverging buffer contents in system call arguments. Hence, the buffer that is passed to the system call could just be compared across variants from within the monitor to find the diverging bytes. If it can be proven that they are pointer values and refer to the same logical allocations, the divergence is benign and can be ignored.

Detecting a pointer value is not as simple as finding a contiguous diverging region of 8 bytes in the buffer though. Some of the 8 bytes in a pointer value might be the same across all variants. For example: the x86-64 architecture requires the most significant 12 bits in a memory address to be the same as the 51st bit [45, p.14], and most implementations extend this to the most significant 16 bits and the 47th bit. As on Linux, only the lower 128TB of address space is actually usable by user-space programs [55], the most significant two bytes in every pointer will be the same across variants. Additionally, some of the other 6 bytes in the pointer might inadvertently match everywhere. Hence, a conservative best-effort approach to reliably ignore benign divergences without falsely ignoring malicious ones is presented here.

Every diverging byte is considered separately. Because the byte is diverging, it must be part of a pointer value, otherwise this is a malicious divergence. Some 8-byte contiguous region that contains this diverging byte must therefore be a pointer value. This defines a

search space or *window* of 8 possibilities as illustrated by Figure 4.1, for which a unified address is constructed and compared across variants. If all the variant's unified addresses match, the investigated 8-byte region was definitely a pointer.

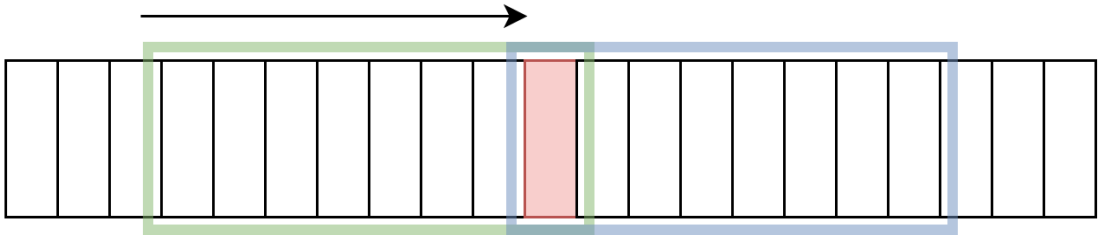


Figure 4.1: All 8-byte contiguous regions that contain a specific byte inside a buffer.

Since this window is only 15 bytes wide, the assumption can be made here that no two full pointers can exist in it next to each other as both require 8 bytes. At most one pointer value can therefore be detected in this window. This is technically false for the following reasons:

1. The least significant $N > 0$ bytes of one pointer could match the most significant N bytes of another pointer and for some reason their matching bytes might overlap in the buffer. Although there is no clear-cut reason for an application to do this, it can technically be constructed. It is not unlikely that the lower bytes of a pointer value match the upper bytes of another though: for a page aligned pointer, the lower byte is 0 and as previously discussed, the upper byte of every user-space pointer will also be 0 on x86-64 in Linux.

2. Without the application's intention and especially when the address space is rather filled up already, which is unlikely on a 64-bit architecture, some other combination inside the window might accidentally be a valid pointer for a specific variant as well. Chances are low though that, even with two variants, the unified values for these accidental pointers match. This could also not be accidental, but rather intended by an attacker as part of some kind of exploit. In that case the MVEE's diversification has been bypassed in some other way already since the attacker was able to figure out equivalent pointer values in all the variants. By only allowing one pointer value to exist in the window though, a divergence can still be raised here.

Deciding how many pointer values may maximally appear in the window is therefore a design choice by the MVEE, balancing application support with security. Due to the high improbability that a false positive, i.e., a non-attacker inserted second pointer value, occurs however, this work advises to require that exactly one pointer value must be found in the window. The buffers are additionally not said to be equivalent if, after finding all pointers, there are still diverging bytes that do not belong to a confirmed pointer.

There exist other pitfalls to this pointer-in-buffer detection though, listed below.

1. Care should be taken to support pointer values into logical allocations that have already been freed. Depending on the implementation of the mapping between variant-specific and unified values, this might not be recognized anymore as a pointer

value since it is no longer valid. Note that this does not necessarily represent a use-after-free bug: when using `write` to write structures to shared mappings as a form of Inter-Process Communication (IPC), the application might be aware of the dangling status of the pointer it is writing. Additionally, it is not the concern of ASB mitigation to predict use-after-free bugs in this way, but rather it is assumed that they will lead to a detected divergence down the line.

2. The start or end of a buffer might be cutting off part of a pointer value, leading to a diverging series of bytes that no pointer can be reconstructed from. This could be mediated by specifying the parts of the 15-byte region that are missing as *wildcards* and matching any pointer value that matches the the parts contained in the buffer, at the cost of decreased security when larger parts of the pointer are missing. This work therefore advises not to cover this edge case due to its assumed limited occurrence.

After applying all four rules for pointer operations and the pointer-in-buffer mitigation, this analysis overextends its purpose since there exist for example pointer to integer conversions that are not unsafe, resulting in security loss and performance overhead. Consider a canonical out of bounds access presented in Listing 4.1.

```
size_t size = 1000;
char* array = (char*) malloc (size);
for (unsigned i = 0; i <= size; i++) {
    printf("ptr: %p \n", array + i);
    *(array + i) = { /* anything */ };
}
```

Listing 4.1: An example of an out-of-bounds array access.

Following the third rule, `array` will be converted to its unified value when calculating the pointer into the array to print, before being deunified to return the eventual pointer value, as shown in Listing 4.2. In case of an out-of-bounds access (`i` = 1000) the result of the sum is now a unified address that points outside the logical allocation. As we will later discuss at length, it is possible that this value does not translate back to an out-of-bounds pointer outside the equivalent variant-specific allocation (`array`). If, at this out-of-bounds address, another logical allocation is mapped in the unified address space, the translation back will assume that this is an in-bounds pointer into that other logical allocation instead. Since the same unified address value will translate to the same logical allocation in all variants by definition, the resulting variant-specific pointer will point to the same logical allocation in all variants. This makes the effect of this out-of-bounds access similar across variants, circumventing MLD.

```
size_t size = 1000;
char* array = (char*) malloc (size);
for (unsigned i = 0; i <= size; i++) {
    printf("ptr: %p \n", deunify(unify(array) + i));
    *((char*)deunify(unify(array) + i)) = { /* anything */ };
}
```

Listing 4.2: An out-of-bounds array access instrumented by considering every operation but dereference expressions unsafe.

This seems like obvious over-mitigation by rule 3, 2 and 1, which consider every addition of integers and pointers to be reconstructions of absolute pointers from relative pointers and deliberately make sure the result is the same logical allocation across variants. Relative pointers are, as discussed in their overview in Chapter 3, not well-defined behaviour and due to reliance on the relative positions of allocations - the very thing MLD is trying to diversify - they seem hard to support from an ASB perspective while retaining maximum diversification. At the point where an integer is added to a pointer value, it is not straightforward to see whether the result is meant to point to another logical allocation, or is the result of accidental out of bounds arithmetic.
Adapting the analysis by removing rule 1, 2 and 3 and not supporting relative pointers does not solve the problem though: the over-mitigation can still occur through rule 4 as evidenced by Listing 4.3, where the iteration simply happens using integers instead of pointers.

```
size_t size = 1000;
std::unordered_set<uintptr_t> set;
uintptr_t array = (uintptr_t) malloc (size);
for (unsigned i = 0; i <= size; i++) {
    uintptr_t array_i = array + i;
    printf("ptr: %p \n", (char*)array_i);
    *((char*)array_i) = { /* anything */ };
    set.insert(array_i);
}
```

Listing 4.3: An out-of-bounds array access using integers instead of pointers.

After instrumentation, an out-of-bounds unified address value may be deunified to the same logical allocation in all variants, again hiding the issue from the MVEE.
Listing 4.3 also shows that this is not some simple edge case that can be detected and accounted for. The integer representations of the pointer values (`array_i`) that are used for the iteration are also inserted into a C++ STL `unordered_set`, using a hash table internally [31, Section 26.2.7]. This highlights how the very same integer can cause a security issue when mitigated *and* strong ASB when unmitigated at the same time. This makes it impossible to define a boundary of only pointer operations to decide whether some value should be unified or not: it is a specific use of the *integer* that is unsafe in this case, namely its insertion into the hash table, not of the pointer itself. However, moving the boundary to include some integer operations does not solve the problem either, as it

is not necessarily clear in the general case which integers come from pointers, and which should have their operations instrumented.

This strategy additionally suffers from the same bug introduction issues as the last. In the `strlen` case from Listing 3.3, both bugs discussed in Section 3.4 could, again, occur if the alignment result does not match the actual alignment of the pointer.

Lastly, an advantage of this approach compared to the previous one is that for guaranteed-safe, bug-free programs, meaning those that never do pointer-to-int conversions and only do arithmetic and comparisons on pointers from the same logical allocations, no instrumentation is performed and none of the secondary metrics are affected.

Using automatic program analysis

Both of the above strategies provide a conservative[2] solution for the problem of ASB, while having substantial security and performance issues. Automatic program analysis can serve as a way of heuristically improving those metrics while still retaining the full ASB coverage that the approaches offer. Note that, contrary to the use of program analysis as discussed at the beginning of this section, there is no reliance on the analysis for the completeness of the ASB mitigation when it is used in this way. It merely improves the secondary metrics with limited precision.

In the case of the pointer-granular analysis, where pointer values exist in unified form *by default*, the automated analysis needs to prove that a certain address value is only used safely for a certain snippet of code. It could then delay the unification of the pointer value until it is passed to an external routine or escapes the investigated snippet otherwise. This allows for the instrumentation of load/store operations to be temporarily disabled in guaranteed-safe circumstances. Within those *safe zones*, there will be no performance overhead or security inhibition. The impact that this has on those metrics depends on the precision of the analysis tool and the amount of unsafe ASB in the program.

[2]The operation-granular strategy does not guarantee complete ASB coverage due to the finite support for conversion possibilities between pointers and other types, but is nonetheless expected to cover a substantial amount of ASB.

```
size_t size = 1000;
std::unordered_set<uintptr_t> set;
// array not instrumented
uintptr_t array = (uintptr_t) malloc (size);
for (unsigned i = 0; i <= size; i++) {
    // array_i is variant-specific
    uintptr_t array_i = array + i;
    printf("ptr: %p \n", (char*)unify(array_i));
    // dereference is guaranteed-safe ASB
    *((char*)array_i) = { /* anything */ };
    set.insert(unify(array_i));
}
```

Listing 4.4: An out-of-bounds array access using integers instead of pointers, instrumented by automatic program analysis based on the pointer-granular approach.

For the operation-granular approach where pointer values are allowed to exist in their variant-specific form until certain operations occur, the automated analysis would try to construct similar *safe zones* where integer values derived from pointer values are not used unsafely, meaning that they are simply used for operations that are considered safe on pointer values and their conversion from a pointer value should therefore not necessarily incur a unification operation.
Note that, in the case of Listing 4.3, the pointer-granular approach could more easily catch this case with the use of automatic program analysis, as shown in Listing 4.4. Assuming that this snippet represents the only use of `array`, it merely serves to construct `array_i` in every iteration. Because the provenance of `i` is very clear in this case, the analysis could rule out that `array + i` is a reconstructed relative pointer across allocations and the variant-specific form of `array` could therefore be allowed to be returned from the `malloc` call. The value of `array_i` is therefore variant-specific as well, and its second use constitutes merely a dereference expression, which is guaranteed safe. For both other uses, i.e. as arguments to `printf` and `std::unordered_set::insert`, no such guarantees exist and it will therefore be unified. This is the ideal case scenario, in which both cases of unsafe ASB are mitigated and no memory errors are hidden as the `array` iteration happens with variant-specific values.
This is not possible for the operation-granular strategy as decides at the type-conversion level whether a value should be (de)unified before conversion. Given the same information as the previous approach, it can not treat the three uses of `array_i` any differently as they come from the same pointer-to-int conversion after the `malloc` call. It will therefore decide to unify during that conversion, and the iteration will happen over unified values.

The pointer-granular approach therefore seems more flexible when combined with automatic analysis tools, even though without it, it performs worse on all secondary metrics even for guaranteed-safe programs. The full scheme for augmenting the pointer-granular approach with automated program analysis is as follows:

1. For every function in which the address of a logical allocation is used as specified in Section 3.4, an intra-procedural analysis tool determines at which point the safety

of the operation can no longer be guaranteed. The type of the pointer value is not considered in this analysis, i.e. pointer-to-int conversions are treated as no-op, unless they truncate. In general, if an operation results in a value that does not represent a memory address, such as the lowest 4 bytes of a pointer value or a relative pointer, the pointer value is unified before that operation. This is equivalent to considering every operation on a derived pointer value that is not supported on an instance of type pointer to be unsafe.
It is also considered an unsafe operation for the pointer value to escape the function that is under analysis.

2. Following step 1, pointer values that are obtained as arguments or that are accessed via global data are guaranteed to be unified. For those, the same analysis is applied to investigate whether they can be temporarily deunified.

3. All functions are investigated again to eliminate redundant pairs of `unify`/`deunify` operations. Essentially, for every function that deunifies an argument before its first use, every unification operation for that argument at every call site of that function can be removed. This is mostly to improve performance.

The approach presented here is a generalization of the concrete analysis performed for Listing 4.3 earlier. However it still falls short in some cases, notably in *glibc*'s `strlen` case from Listing 3.3: as the value obtained from the alignment check does not represent a memory address, this analysis will unify the pointer value before the alignment check, potentially resulting in both known bugs, if the bogus value is chosen poorly.

4.2.2 Designing the unified address space

In the above section, the existence of a unified address space was assumed to and from which variant-specific pointer values could be freely translated. This section explores the design of such an address space while gradually uncovering the implicit requirements placed on it by the mitigation strategies.

Magic number

The most naive way to come up with a variant-agnostic value as input to an address-sensitive operation is to just define a magic number up front and use that every time. Surprisingly, this works without problem in the hash table case: since it is non-reversed strong ASB, any value could be provided for it during the hash operation, including the same one for every pointer. As discussed in Chapter 3, this will not compromise the table's functionality. Similarly, the case where pointer values are converted to a string representation to be printed out is also trivially handled by this mapping.
The solution breaks down however whenever there exists the need for either validating that two pointers refer to the same logical allocation in multiple variants or when there is some kind of feedback from the unified value back to the original pointer value. For example, in the relative pointers example from Listing 3.1, all the reconstructed `abs_ptrs` will have the same value regardless of what their original value was. This could be an invalid memory address depending on how the magic number is chosen or how the pointer subtraction operation is handled.

The need presents itself for a unified address space where all pointer values have different unified values.

Count-by-use

A less naive way of mapping address-dependent values would be to provide a new unique value for every address-sensitive operation. This could be done by simply incrementing a counter on every instance of such an operation, returning the counter value and keeping track of the values that were already returned in a cache, so the same unified value can be returned for the same address-dependent value in the future again. As the variants execute the same code, every address-sensitive operation happens in the same order and the unified values will be consistent in the variants.
This still supports non-reversed strong ASB, but is not able to definitively validate pointer arguments or pointers in buffers as it will return a new unified address for any pointer value it has not unified before, bearing no validation that it refers to the same logical allocation in all variants. If the program writes `struct` contents to a file containing a pointer value, an attacker might try to overwrite the pointer value in one of the variants with sensitive data, to leak it. The MVEE should be able to validate that this (1) is not a pointer value and (2) is not an equivalent value to the other variants. Using this mapping, any value will simply yield an equivalent unified value, with no guarantee that the variant-specific values are equivalent. Therefore, the mapping should at least be able to validate that a value is a pointer, for the pointer in buffer case.

The advantage of this approach is that *any* value tainted by an address can be mapped, not just a pointer value. Although not investigated in detail as a detection strategy due to its assumed infeasibility, the suggestion at the end of Chapter 3 that some cases of reversed strong ASB could be reduced to non-reversed strong ASB is a use case for this property: the relative pointers example, analyzed there, would require a mitigation strategy to provide bogus values during the sort operation. However the relative pointers are not valid addresses, so a mapping is required that supports any kind of value. This mapping fulfills that requirement.
As previously discussed, generalizing this approach of finding a point during any strong address-sensitive operation at which there is no feedback to the functionality of the program is no small feat and deemed impossible in the general case by this work. The following mappings will therefore not concern themselves with supporting non-pointer values, and instead focus on fixing the issues this mapping has, notably with the unreliable verification of pointer values.

Count-by-allocation

Similarly to how every address-sensitive operation happens in the same order in all the variants, every logical allocation occurs in the same order as well and can thus be identified in a variant-agnostic way through their position in the occurrence order, from here on referred to as *occurrence index*. If all logical allocations were to be registered, an *occurrence list* could then be kept that contains the base pointer for every logical allocation in the order in which they were allocated. Although this is the easiest to imagine for dynamic allocations, other sources of valid pointer values covered in Section 3.4 also fol-

low a variant-agnostic logical allocation order - for stack allocations the order of function frames being pushed or `alloca` being called, for globals an arbitrary order defined by the implementation that instruments the code to register them. When the logical allocations are freed via function exit or call to `free`, they are consequently *deregistered* from the occurrence list. Note that this does not necessarily mean that the entry is removed: for some mitigation strategies, it might be required that occurrence indices are never re-used. The deregistration procedure is therefore mapping- and strategy-dependent.

This mapping supports translating variant-specific offset pointers into the unified address-space, namely by finding the logical allocation whose base pointer is the closest to but still smaller than the offset pointer. Since we assume that logical allocations can never overlap, this is guaranteed to be the logical allocation containing the offset pointer. The unified value for the offset pointer is then the result of adding the offset into the logical allocation to the unified value of the base pointer. This mapping can therefore be used to validate any pointer value, as the occurrence index of its logical allocation and its offset into that logical allocation represent a combination of two numbers that is unique to that pointer value.

However, when unified pointers undergo arithmetic such as in the example from Listing 4.3, they should definitely not be deunified to a different logical allocation in all variants when the arithmetic performed on them is still within bounds ($i < 1000$). Yet this is what happens for the occurrence list mapping: `unify(x)+5` will be either a valid occurrence index for a different logical allocation or not contained in the occurrence list at all. Either way, the functionality of the program is compromised as the correct result, which is an offset pointer into the original logical allocations, is not obtained.
Therefore, this mapping can be regarded as a many-to-one mapping from the Variant-Specific Address Space (VSAS) to the Unified Address Space (UAS): every variant-specific pointer can be mapped to a single unified address, but a unified address can be the unification of one offset pointer for any logical allocation in the occurrence list. If there are N entries in the occurrence list, there are N variant-specific values that produce this unified value.

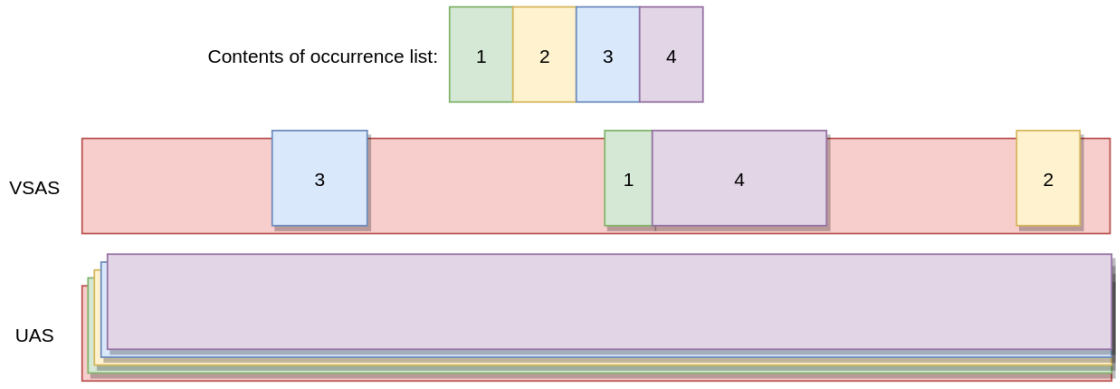


Figure 4.2: Layout of VSAS and UAS in a mapping without size information.

Translating from the variant-specific to the unified address space is $O(n)$ since every entry has to be investigated to find the biggest base pointer smaller than the unified pointer.

Going in reverse is not possible due to the many-to-one nature, which is a big shortcoming of this mapping and introduces bugs.

Count-by-allocation with full size information

If the size of the logical allocation were to be stored alongside its base pointer in the occurrence list, the unified value for any base pointer value could be found by summing up the sizes of all previous allocations. For offset pointers, a similar strategy to the previous many-to-one mapping can be employed by adding the offset to the unified value of the base pointer, but since there is now just as much space *reserved* in the unified address space as there are valid offsets into the allocation, the unified values of the offset pointers will be unique to that offset pointer. This provides a one-to-one mapping from theUAS to the VSAS: there are as many unified addresses as there are variant-specific ones.
Note that it is transparent to the functionality whether the size of the logical allocation is stored, or its unified value. If the former is used, the unified value can be found as described above by traversing the list. If case the latter is stored the size can be found by subtracting this allocation's unified value from the next entry's as they will be spaced apart by its size.

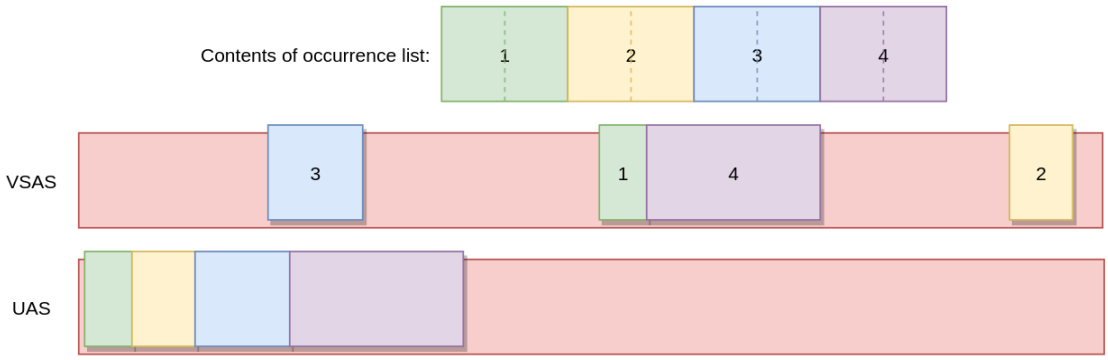


Figure 4.3: Layout of VSAS and UAS in a mapping with full size information.

Because there are exactly as many unified addresses as there are variant-specific ones, there is also no overlap between logical allocations in the unified address space. Hence, it can be conclusively decided from a given entry in the occurrence list whether it contains a pointer or not, eradicating the need to traverse the whole list for every unification procedure. Regardless of whether the size or the unified address is stored, this is basically a search into an unsorted list rendering it $O(n/2)$ on average. For a deunification procedure though, time-complexity gains can be achieved by storing the unified addresses directly instead. If the size were stored, the list needs to be traversed linearly while computing the unified values until an entry is found whose unified address range contains the value to deunify. This is again $O(n/2)$. However, if the unified addresses are stored in the first place, the list is sorted by definition. Hence, a binary search algorithm could be employed to achieve $O(\log(n))$. The time complexity of this algorithm can therefore be improved by storing the logical allocations variant-specific and unified address together.

The main downside of this approach is its memory overhead: it is twice that of the one

without size information[3]. The next approach aims to mitigate this memory overhead while maximally preserving the time-complexity advantages of this mapping.

Segmented count-by-allocation with aligned size information

By aligning and rounding up the size of the logical allocation[4] that is being inserted into the occurrence list to a certain predefined boundary or *segment size*, and then storing an entry into the occurrence list for every *segment* the logical allocation occupies, the occurrence index can be calculated similarly to the above one-to-one mapping. However, since the segment size is constant, the size of the allocation does not need to be stored anymore. This also provides a one-to-one mapping from the variant-specific values to the unified addresses, but has, depending on the chosen segment size, a smaller memory overhead as illustrated by a comparison between Figure 4.3 and Figure 4.4: although the latter has to store more entries, they are half the size of the former, causing the total bytes stored to be lower in this example (`2*4*sizeof(void*)` vs `7*sizeof(void*)`).

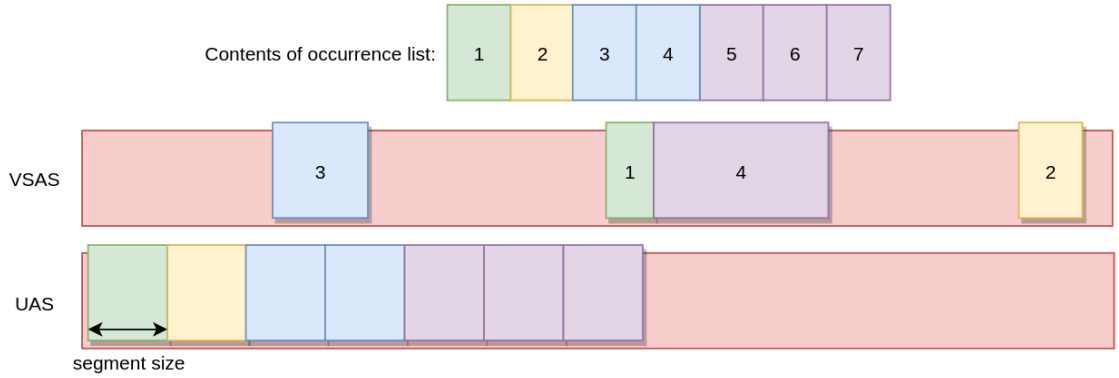


Figure 4.4: Layout of VSAS and UAS in a segmented mapping with aligned size information.

Due to the alignment operation, the property of the previous mapping that there are as many unified addresses as there are variant specific ones is lost; there are more unified addresses now. They are located in the remaining part of the last segment of an allocation and they map the difference between the aligned up size and the actual size. This means that for a given pointer value, there might be multiple segment entries that seem to contain it, similarly to the mapping without size information. Consider the first (green) and fourth (purple) allocation in Figure 4.4, which sit right next to each other in the VSAS. The entry that is stored for the green allocation is bigger than its actual size because it is aligned up to the segment size, meaning that the following operations happen during the unification of the purple base pointer: the occurrence list is walked, and for entry entry, it is checked whether this entry contains the variant-specific base pointer. The first entry that is investigated is the green entry, which claims to span a region as large as the segment size in the variant-specific address space. If we look at the VSAS, this means it claims that its allocation covers a small part of the purple allocation as well. We might

[3]Note that if upper bounds are assumed by the application for either the maximum allocation size or the maximum memory usage, storing the size or the unified address respectively could have memory efficiency advantages.

[4]Only the size that is being inserted, not the actual allocation size.

therefore think that the pointer we are unifying is a green pointer, while it is not. To make sure the purple unified address is returned instead, the entire occurrence list has to be traversed again to find the entry with the highest base pointer that is smaller than the pointer to unify. This is the only way to distinguish between all the allocations that seemingly claim to contain a pointer value: the allocation allocation that contains it will have the highest base pointer value of them all, as allocations in the VSAS can not overlap. Therefore, unification is $O(n)$ again. For deunification, the property of this mapping that every segment is the same size can be exploited: the index in the occurrence list where the segment containing the unified value resides can be found by integer division between the unified value and the segment size, since the unified base pointers per segment will always be a multiple of the segment size. If the occurrence list has random access, the deunification can therefore be $O(1)$.

Note that there exists a trade-off between wasted unified address space and memory overhead when choosing the segment size. This is illustrated by the difference in UAS usage between Figure 4.3 and 4.4. The bigger the segment size, the larger the *wasted* areas of segments will be that map the difference between the aligned size and the actual size, but the fewer segments and therefore occurrence list entries are needed to represent logical allocations. In a 64-bit environment the unified addresses are 64-bit, making their exhaustion less of a concern. Additionally, since the base pointers are also 8 bytes wide, the memory overhead to store a segment entry is twice as big as in a 32-bit environment. In 64-bit, larger segment sizes might therefore be preferred.

Care must be taken though to support logical allocations using `mmap` that have the unique property that the pages they return within one mapping are contiguous, but can be individually unmapped. Consider for example a segment size of 16KB, a page size of 4KB and an `mmap` call for 3 contiguous pages, after which only the middle page is unmapped. A naive solution would be to map every page in a separate segment such that the middle segment could be promptly deregistered, but this would violate the programmers expectation that the pages are contiguous in the address space: there will be 12K unified addresses between the end of the first page and the start of the second. Additionally, the difference between the location of the third and first pages is not 4K as expected, but 16K in the unified address space.

Alternatively, only a single segment could be allocated for the entire 12KB region, but it is unclear how the `munmap` call for the middle page can then be handled. A simple solution is to make sure that the page size is a multiple of the segment size, meaning that every page will be fully contained in $M \geq 1$ segments. Every single page can then be trivially unmapped by deregistering all the segments it spans. This upper bounds the segment size to the page size however, which may be too small sometimes to maximally benefit from other optimizations. For example, to improve the unification performance of this segmented mapping, it could be altered to additionally store the size of the logical allocation in the segment entry. For some segments, this would be the segment size, but for others this could store the size of the remainder of the logical allocation that could not quite fill up the whole segment. This gains back the property for every segment to be able to decide conclusively whether a variant-specific pointer is contained in it or not, halving the amount of entries that need to be traversed in a unification operation on average. To combat the memory overhead, the stored base pointers could be *tagged* with the size information instead of storing a separate integer of metadata. On the x86-64 architecture the most significant 12 bits of a pointer value are essentially unused [45, p.14], allowing

arbitrary storage there[5] without increasing the memory overhead of the segment entry. These 12 bits exactly match the 4KB page size[6]. In practice however, most current x86-64 implementations do not use more than 48 bits[7] of that address space, allowing a 16-bit value to be stored inside a pointer[8]. Segment sizes of up to 64KB could therefore be technically supported, further decreasing the memory overhead of the occurrence list by storing even bigger allocations in a less entries, but the previously discussed use of `mmap` restricts it to just 4KB.

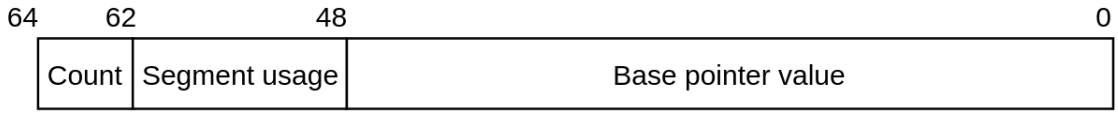


Figure 4.5: Layout of an 8-byte segment entry in a mapping with segment size 16KB and `mmap` support.

A superior solution is to make the segment size 16KB using 14 out of 16 most significant bits, and then using the remaining 2 bits to count the amount of pages that have already been unmapped from the segment. The resulting layout of the pointer value is illustrated by Figure 4.5. As 4 pages fit into a single 16KB segment, this perfectly matches the storage provided by the 2 remaining bits. The segment will not be deregistered until all four pages contained in it are. None of the pages should be actually unmapped until the program has `munmap`'ed them all. Rather, they should be marked `PROT_NONE` using `mprotect` [56]. Otherwise, the OS might inadvertently return the page to some variants on another `mmap` call down the line, causing the same variant-specific page to exist in multiple registered allocations at the same time. This could lead to different variants returning different unified values for the same logical allocation, potentially resulting in unsafe ASB. If the pages are never returned to the OS in the first place, this can never happen. Using `mprotect` with `PROT_NONE` preserves the expected behaviour of an unmapped page by causing a fault on every access. To prevent the page from occupying memory and counteracting all memory efficiency optimizations that were just presented, it could also marked as unneeded using `madvise` with `MADV_DONTNEED` [57], allowing the OS to reclaim the physical memory.

Note that the expected memory efficiency gains of using 16KB segments are small, as suggested by the limited amount of dynamic allocations that are bigger than 4KB in real-world applications [58]. One could therefore wonder if the added memory efficiency is worth the increased implementation complexity in some cases.

In conclusion, the optimized implementation of this segmented mapping that stores how full the segments are in the base pointer values has an amortized 8-byte overhead[9] per

[5] As long as the most significant bits are set back to the value of the 51st bit on a deunification operation.

[6] We disregard the use of huge pages in this work.

[7] As previously discussed, user-space programs in 64-bit Linux only use the lower 128TB or address space. Therefore, the upper 18 bits of every pointer value will actually be 0 in user-space, allowing even greater memory efficiency. In the analysis of every mapping, only 16-bit is used though to illustrate the concepts and keep a greater generality across different operation systems.

[8] As long as the most significant bits are set back to the value of the 47th bit on a deunification operation.

[9] Assuming that the vast majority of logical allocations, including static ones, are smaller than 16/64KB.

logical allocation, $O(n/2)$ unification complexity and $O(1)$ deunification complexity. This outperforms the previous mapping in terms of memory and run-time overhead.

Spaced out mapping with fixed padding

All the mappings presented up until now still share the security flaw highlighted by the out-of-bounds access in Listing 4.3, where, depending on the detection strategy, the iteration will happen in unified space and the $i = 1000$ access will deunify to the same logical allocation in all variants. To stochastically minimize the poor effect this has of potentially hiding true positive divergences, the logical allocations could be spaced out across the unified address space with *free* or *unregistered* regions in between. This way, it is detected when a value deunifies to such an unregistered region, and appropriate action can be taken. This includes either returning `NULL` or a different pointer value in all the variants. Another possibility is to just terminate execution, as it has been detected that the variants are trying to obtain a pointer value that does not refer to a valid place inside a logical allocation. All of these would be rather premature precautions however, as the memory needs to be accessed prior for it to be an exploitable error. In C11, as soon as a pointer value is obtained using this arithmetic that does not point into an *array object* or one past the end of the *array object*, it is explicitly undefined behaviour [21, Section 6.5.6.8]. In C++17, it depends on whether the implementation enforces *strict pointer safety* or *relaxed pointer safety* [31, Section 6.7.4.3.4]. For strict pointer safety, the use of any pointer that is not *safely derived* is undefined behaviour. For relaxed pointer safety, the validity of a pointer does not depend on whether it was safely derived, as long as it compares equal to some safely derived pointer value its use is not necessarily undefined behaviour. Although the out of bounds access in Listing 4.3 is clearly undefined behaviour for both languages and anything could be returned as long as it is not the same in the variants or causes a fault, the expected behaviour of the code in Listing 4.5 is that the dereference of `ptr` is not an invalid access, and the 500th byte into the allocation should be an `'a'`.

```
uintptr_t i_ptr = (uintptr_t) unify(malloc(1000));
// ... no modification to i_ptr
i_ptr += 1500;
printf("%p\n", i_ptr);
char* ptr = (char*) deunify(i_ptr);
ptr -= 1000;
*ptr = 'a';
```

Listing 4.5: Deunification of an out-of-bounds unified value.

Suppose that the result of the call to `malloc` gets immediately unified[10] and that during the type conversion to **`char`**`*`, a deunification operation happens.
In the count-by-allocation mapping with full size information presented above, the offset unified value will point into a different logical allocation, and that one's base pointer will be used for the deunification operation, leading to a faulty result, consistent in all variants. In the segmented mapping with a segment size bigger than 1500, the base pointer of the

[10] In this simple case, it likely would not if any kind of automated analysis is used, but this is assumed here for the sake of discussion.

logical allocation made by `malloc` will be used instead, and the result of the `deunify` call will be an out of bounds **`char`**$\star$ 500 past the end of the allocation, similarly to how `i_ptr` is a unified out-of-bounds pointer 500 past the end of the allocation in the UAS. However if the out-of-bounds access was even further from the allocation or the segment size was smaller, the same result as in the mapping with full size information could have been obtained.
It therefore seems like over-estimating the size of the allocation as naturally done by the segmented mapping actually helps to mitigate the security issues caused by the mapping until now. This makes sense: if there existed only one logical allocation and all pointers were derived from it, arbitrary arithmetic spanning the full 64-bit address space could be performed on both its unified and variant-specific value without ever deunifying to a different logical allocation. It is therefore key to provide as much *padding* as possible around every logical allocation such that out-of-bounds deunifications consider the correct base pointer as theirs and when the pointer is brought back in bounds or an out-of-bounds access happens, the results are consistent with the non-instrumented version of the variant. There is no way to provide every logical allocation with the full 64-bit range of supported arithmetic in the unified address space it should theoretically get to guarantee that no out-of-bounds deunification will lead to the same faulty logical allocation in all the variants. Instead, a best-effort spacing of the logical allocations within a single 64-bit address space[11] is attempted. The main assumption here is that out-of-bounds accesses are more likely to happen closely around the logical allocation than somewhere further away.

A simple way of achieving padding between logical allocations in the unified address space is by assuming a maximum number and size of allocations up front, say both 2^{32}. A fixed buffer of 4GB can then be assumed around every logical allocation[12], catching all underflows and overflows within that region, depending on where the base pointer of the logical allocation is placed. If placed in the middle, 2GB of overflow is caught and 2GB of underflow. This is essentially the same as a segmented mapping without storing size information and a segment size of 4GB. The earlier count strategy could again be employed to support `mmap`, *with* memory overhead in this case; there can exist up to a million contiguous pages inside every 4GB logical allocation, which is more than the unused most significant bits in a tagged pointer can hold.

[11] The possibility of a *fat pointer* is not investigated in this work. Although an increased address space (>64-bit) would allow for more spaced out placement of the logical allocations, it does not solve the fundamental issue of unified arithmetic resulting in a deunification based on a different logical allocation. The implementation complexity and memory overhead associated with increasing both native pointer widths and integer widths is also non-negligible.

[12] Contrary to other red zone or electric fence schemes, we do not have to worry about the additional memory consumption of our padding zones. The UAS is a made-up concept.

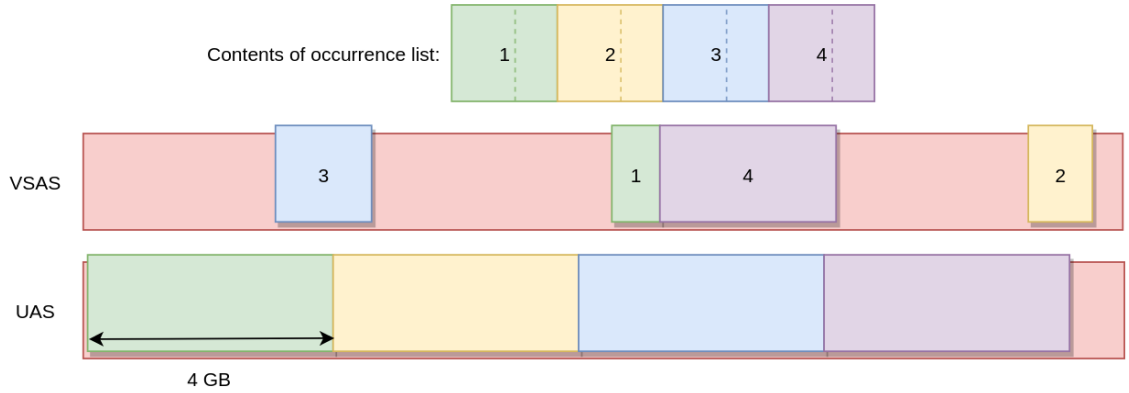


Figure 4.6: Layout of VSAS and UAS in a spaced out mapping with fixed 4GB padding.

Unification becomes $O(n)$ again unless size information is additionally stored. Deunification is $O(1)$ again, as the index into the mapping can straightforwardly be read from the most significant 32 bit of the unified value.

Spaced out mapping with dynamic padding

To support `mmap` or to improve unification performance, the memory overhead of the spaced out mapping with fixed padding increases rapidly: the count value requires at least 20 bits of storage unless upper bounds are assumed for that as well, and the size is upper bounded at an additional 32 bits of storage. Also, although generous, the assumptions it makes about the size of logical allocations and the amount of logical allocations do not necessarily hold for every application. If they underestimate the actual amount, the potential of having even more padding surrounding the logical allocations is wasted. The representation in Figure 4.6 is merely conceptual and greatly overstates the coverage of the UAS by the logical allocations in this regard. In reality, these four allocations occupy only $4 * 2^{32}/2^{64} = 2.33\text{e}{-8}\%$ of the UAS.
The need presents itself for a more flexible approach where all of the available address space is always used to provide padding and the chances of faulty deunifications are minimized to a theoretical minimum within the 64-bit address range. To this end, an iteration over all 64-bit unified addresses is needed that visits every address exactly once but does so in the most spaced out way possible: the middle address is visited first, then an address a quarter into the address space, three-quarters into it, etc. This can be done by considering a hypothetical balanced binary tree of all 64-bit values. A breadth-first (*level-order*) traversal of that tree will visit every value in the desired order, as illustrated by Figure 4.7. This is referred to as the *spaced out iteration order* of the address space.

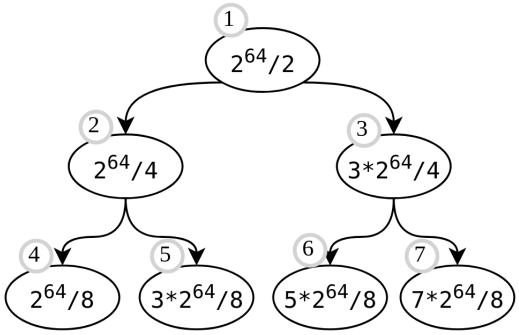


Figure 4.7: Breadth-first search through a balanced binary tree of all 64-bit numbers.

If every logical allocation, regardless of its size, is now appointed to the next unified address in the spaced out iteration order, the unallocated parts of the address space are always maximally used to provide padding around the allocated regions, as can be seen in Figure 4.8.

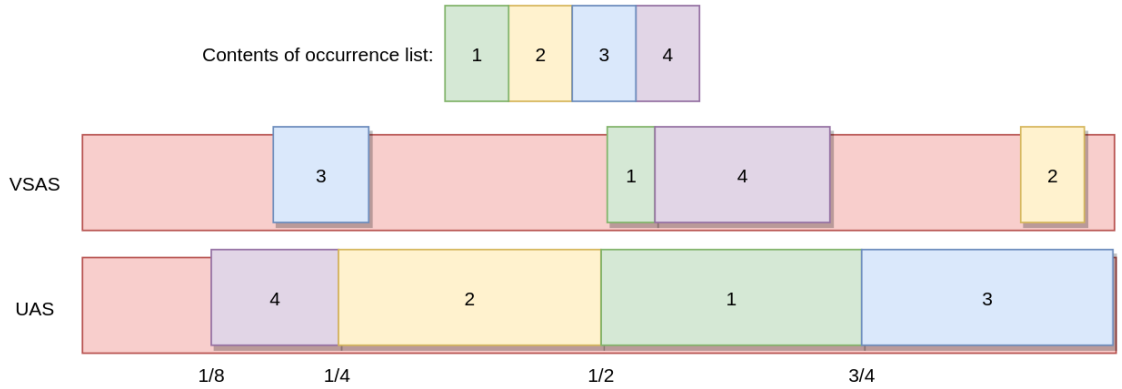


Figure 4.8: Layout of VSAS and UAS in a spaced out mapping with dynamic padding.

Note that he representation of the VSAS and UAS has not been realistically scaled relative to the size of the allocations in these figures for the sake of clarity. In this case, although the purple allocation is smaller in the UAS than it is in the VSAS, it is rather unlikely that the actual size of the purple allocation would actually be bigger than $2^{64}/8$, or around 2.3 quintillion bytes[13], which is the difference between the unified base pointers of the yellow and purple allocations. This is how big it would need to be in order to be limited in size by its storage in the UAS for this amount of logical allocations. Nevertheless it does highlight how this mapping does not take into account the size of the allocations at all, and assumes just that they will fit. This is not necessarily a problematic assumption: a contiguous allocation of 4GB is only compromised as soon as allocations are stored next to each other separated by a mere 4GB of padding, which only happens 32 levels deep into the tree. To reach this level, $\sum_{n=1}^{31}(2^{n-1}) = 2^{31} - 1$, or roughly 2 billion logical allocations need to be registered[14]. At that point however, it merely becomes a theoretical possibility.

[13] In fact, it is impossible in practice given the architecturally defined addressable limit of 48 bits on x86-64.

[14] Note that these allocations can be any size. Every allocation occupies an entry in the occurrence list.

Even more allocations need to happen to increase the chances of collisions between logical allocations. Although not impossible, this work is not immediately concerned with the exhaustion of dynamic padding.

A unification operation walks the occurrence list and associates every entry with its corresponding unified address according to the spaced out iteration order. It determines the logical allocation with the highest base pointer that is still smaller than the pointer to unify and bases its result off of that entry. Note that optionally, the size of the allocation can be stored (either segmented or not) to reduce the amount of entries that need to be walked if the pointer is in bounds. For out of bounds pointers, the entire list still needs to be walked to find the appropriate logical allocation.
A deunification operation needs to find the entry in the occurrence list that has the biggest unified base pointer still smaller than the value to deunify. Similarly to the unification operation, if the size of the allocation is stored the complexity can be reduced to $O(n/2)$ for in-bounds values.

To improve the performance of unification and deunification operations, the spaced out iteration order can be investigated in more detail. Essentially, this mapping relies on the fact that for an occurrence list with size N, the $\lceil \log_2(N+1) \rceil$ most significant bits in every unified address uniquely identify a logical allocation and that the least significant bits represent the offset into that allocation. Hence, there exists some non-random one-to-one mapping from the most significant bits in a unified address to an index in the occurrence list, i.e. the level-order index of an entry in the tree. The most significant bits of the unified addresses essentially follow the following pattern in the spaced out iteration order: `1...`, `01...`, `11...`, `001...`, `101...`, etc. This is a simple big-endian binary counting sequence. In fact, when reversing the endianness of the $\lceil \log_2(N+1) \rceil$ most significant bits in a unified address, the one-based level-order indices of the logical allocations, shown in Figure 4.7, are obtained. The occurrence list stores logical allocations in the level-order of their unified base pointers, meaning that these most significant bits can trivially be translated into an index in the occurrence list, reducing the complexity of deunification operations to $O(1)$.

To support `mmap`, the counting method presented in the segmented mapping can be employed again. If entries are restricted to contain the base pointer only (no size information), there are 16 bits in every entry that are unused and in which the count of pages can be stored. This provides a maximum of over 64K pages, or 256MB. The spaced out mapping is merely an efficient and more secure way of iterating the unified address space though and the metadata that is stored in the occurrence list can still be of any layout and size. For example, if only 15 out of 16 bits are used to store the count value (still providing 128MB), the extra bit can be used to signify whether the base pointer points directly to a logical allocation, or to some metadata struct that then indirectly refers to the logical allocation, besides providing additional count storage. This limits the memory overhead of supporting excessively large `mmap`'ed regions to only those allocations that require it.

This mapping theoretically outperforms every previous mapping in terms of memory overhead and unification/deunification performance, while additionally providing the best sup-

port for out-of-bounds pointers. It provides linear unification complexity and constant-time deunification complexity. It can also provide constant-time registration complexity, if a larger memory overhead is acceptable, by never re-using an entry in the occurrence list.

Complying with the alignment expectations of address values

At the beginning of Chapter 3 we explained our assumption that generally no guarantees can hold about the addresses of logical allocations in the variants, except when such guarantees are explicitly provided by an API. Most notably, these include alignment guarantees such as by calls to `aligned_alloc`. Additionally, `malloc` is also guaranteed to align the memory it allocates to a boundary that is suitable for any object [21, Section 7.22.3.1]. Although the alignment requirements of objects are technically implementation-defined [21, Section 6.2.8], most implementations recursively define it as the largest alignment requirements of its members, so in essence it is never greater than the largest alignment requirement of a fundamental type, which is `alignof(max_align_t)` [21, Section 6.2.8.2]. This is typically either 8 or 16 bytes.

The unified addresses representing these allocations should satisfy the same alignment guarantees, trivially exemplified by the snippet in listing 4.6.

```
void my_api_call(void* p) {
    // p should be page aligned!!
    assert(!((uintptr_t)p & 0xFFFU));
    // ...
}
```

Listing 4.6: An example of an alignment check that forces unified addresses to have similar alignment to variant-specific ones.

All valid alignment boundaries are non-negative, integral powers of 2 [21, Section 6.2.8.4]. Hence, a value may accidentally also be aligned to a larger boundary than intended, because the set of values that are aligned to a boundary 2^B is a strict subset of the set of values that are aligned to a boundary $2^b < 2^B$. In other words: all values $K*2^B$ that satisfy an alignment 2^B also necessarily satisfy every alignment $2^b < 2^B$ since 2^B is divisible by 2^b. If a pointer value is obtained in one variant with a specific alignment, there might exist a number of greater, *accidental* boundaries, that the equivalent pointers in the other variants are not aligned to because it was not part of the API guarantees. The value of a pointer will therefore satisfy a non-consistent amount of alignment boundaries across the variants in the general case. This introduces non-determinism into the program: the result of an alignment check on a pointer value to a boundary greater than the boundary that pointer value was aligned to is essentially random in the variants. We can not simply solve this by merely providing a unified address that satisfies the API-guaranteed alignment: even that might have greater, accidental boundaries it is also aligned to. Although the result of the alignment check will always be consistent in the variants, some of the variants' pointers may not satisfy the alignment that the unified address satisfied. It is therefore important that the unified address is not aligned to any greater boundaries than the one that is

API-specified. More generally, the unified address may not be aligned to any boundaries that not all variant-specific values are aligned to (i).

In addition, we can not simply make sure that the unified value will just not satisfy any alignment checks to avoid the problem. Apart from the fact that an application might not expect an alignment check to always fail, it is also simply impossible. We could try to unify every base pointer to an odd number, but offset pointers into the allocation could still be even, or aligned to an even greater boundary. This highlights how not only the base pointers' unified values should be suitably aligned, but also all of the offset pointers. The only way to do this is by making the unified base pointer at least aligned to all of the boundaries that the variants' pointers are aligned to for that allocation (ii).

(i) and (ii) restrict the unified base pointer to be aligned to exactly as many boundaries as all of the variant's pointers are. This is equivalent to being aligned to the biggest boundary that all of the variant's pointers are aligned to.

The biggest boundary that a value is aligned to is 2^N where N is the number of times the value is divisible by 2. From a binary perspective, N is the number of trailing zero bits in the value. Hence, to ensure that the value does not satisfy any greater boundaries than 2^N, the Nth bit in the binary representation of *value* should be set. Although this ensures that the unified base pointer P to an allocation does not cause any issues in an alignment check, also all of the offset pointers into the allocation should similarly not satisfy any alignment checks that not all variants will satisfy. An example of this is *glibc*'s `malloc` implementation, discussed in Chapter 3. The location of the 1MB aligned pointer inside the 2MB region differed due to the offset pointers into the region having different maximal alignment boundaries in the variants.

To mitigate this, all offset pointers into an allocation must have a similar alignment in all variants and in the unified address space. As Figure 4.9 indicates, this is not the case by default for an aligned allocation.

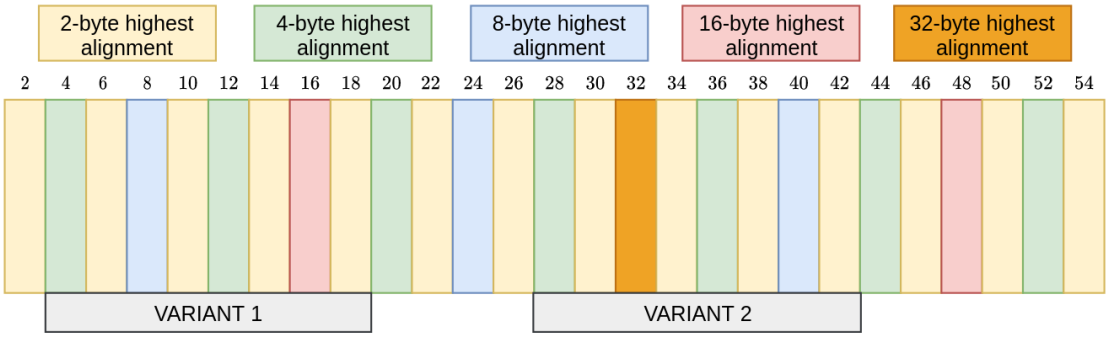


Figure 4.9: Distribution of highest alignment boundaries per address over the address space.

Any region of memory that is aligned to a boundary 2^N is guaranteed to contain the same consecutive highest-alignment boundaries, or *alignment pattern*, for a region of 2^N bytes from its aligned base address. If the size of the allocation, and therefore the range of addresses that alignment checks are supposed to be supported on, extends beyond those first 2^N bytes, the alignment pattern in the rest of the allocation is not guaranteed to be the same across variants. Figure 4.9 shows this for a logical allocation of 16 bytes, aligned

to a 2-byte boundary. Hence, the alignment of a region of memory is not solely defined by the lower $N+1$ bits in the base address, but also by some number of bits on the more significant side of the Nth bit. Every value of those more significant bits yields a different alignment pattern in the allocation because it will determine what happens when a pointer into the allocation is incremented beyond the first boundary and thus beyond the Nth bit.

A naive solution would be to find a pattern alignment value that is guaranteed to always underestimate the alignment of every offset pointer. This is clearly not possible: an allocation of 16 bytes is guaranteed to contain an 8-byte aligned value, whatever the alignment of the base address may be. There is no way to construct two different alignment patterns in which the highest alignment of some address will not be greater than the highest alignment of the corresponding address in the other pattern. The alignment patterns must therefore be equal, both in the variants' address space and in the unified address space.

The bits that define the alignment pattern are the bits that an iteration through the entire allocation accesses. The minimum amount can be expressed as

$$C = \left\lceil \log_2 \left(S + 2^N \right) \right\rceil$$

where S is the size of the allocation and 2^N is the alignment boundary. These C least-significant bits have to be the same in all variants and may never all be zero, as the amount of trailing zeroes will then be determined by the rest of the occurrence index and the variant-specific values might not match that. A simple way to ensure this is by setting all bits to 0 in the base pointer value except the Nth bit. This essentially initializes them to 2^N, after which sufficient bits are reserved to count from 2^N to $2^N + S$. If any of the less significant bits were 1, the base pointer would not satisfy the alignment requirements. If any of the more significant bits (up to bit $C-1$) would be 1, offsets up to S bytes into the allocation would make the amount of reserved bits overflow, temporarily setting all of them to zero. Together, these bits form a kind of mold that all the variants have to fit in to have the same alignment, and we therefore refer to them as the *corset* of the address value. Since the bits to the left of the Nth bit *trap* overflows of the Nth by being 0, we call them the *trap bits*. Figure 4.10 shows the memory layout of a unified base pointer value with the alignment mitigation applied.

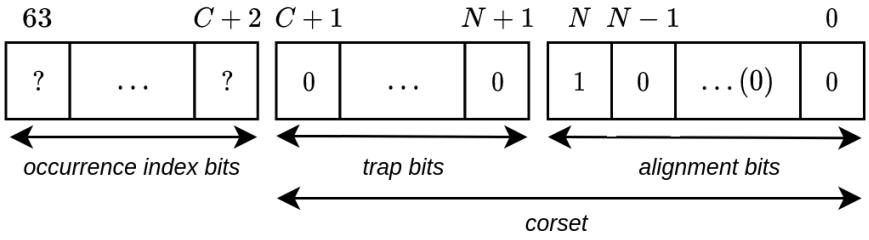


Figure 4.10: Layout of a unified base pointer value.

This amount of trap bits only provides deterministic pattern alignment for up to $2^{C+1} - 2^N$ addresses after the base pointer address. Although this includes the size of the allocation, out-of-bounds pointers might end up in alignment checks as well. It is impossible to fully support alignment checks on out-of-bounds pointers. This is trivially provable by

considering an out-of-bounds pointer F to an allocation with base pointer P and alignment 2^N. To support any F, P would need enough trap bits to cover the entire address range of 2^{64} values, which is

$$C - N - 1 = \left\lceil \log_2 \left(2^6 4 + 2^N\right) \right\rceil - N - 1 = 65 - N - 1 = 64 - N$$

trap bits. Together with the $N + 1$ lower bits that need to be set to 2^N, this makes for a total of $64 - N + N + 1 = 65$ bits that need be set to a fixed value in P. This is an impossible situation for the Nth bit: according to the $64 - N$ most significant bits it should be 0, but according to the $N + 1$ lower bits it should be 1.

In general, the more out-of-bounds pointers are supported around a given allocation, the less logical allocations can be mapped as the trap bits occupy more of the address value. Not supporting alignment checks on arbitrary out-of-bounds accesses might not be a huge problem in practice though since they occur less frequently and are more indicative of a potential memory error than out-of-bounds pointer arithmetic. At the cost of a decreased address space, some redundant trap bits could be used if necessary.

The solution for alignment checks we propose is compatible with all of the presented mappings, potentially requiring them to occupy a slightly bigger portion of the unified address space to ensure that a suitably aligned unified value can be secured for the base pointer. It is worth pointing out how the restrictions placed on unified values to support alignment checks counter-act the spacing of the addresses to support out-of-bounds arithmetic: consider the common situation where at first, a lot of small allocations are made with relatively small alignment requirements. To maximally space out these allocations, we will use the most significant bits in the unified addresses first. Then, suppose some bigger logical allocations with stricter alignment requirements show up. More of the least significant bits will have to be fixed to form the alignment corset in the unified base pointers, and fewer most significant bits will be available to space out the allocations. In fact, using the most significant bits first for the small allocations was the worst possible decision from an alignment perspective. If it were known beforehand that allocations with bigger corsets would show up later, the spacing of the initial small allocations could have been achieved with less significant bits, potentially even using bits that are covered by the bigger allocations' corsets later on anyway. That way, all of the more significant bits remain open to space out the bigger allocations. Conversely, this is the worst possible scenario from a spaced out perspective: the small allocations will not be spread out across the entire address space, unnecessarily wasting padding and increasing the chances of hiding true divergences. It is not clear what the ideal practical balance would be for the average program, but erring on the side of maximal spacing at all times currently seems desirable for the following reasons:

1. It is unclear how to determine for the initial allocations how far they should be spaced apart if maximal spacing is not employed.

2. For every allocation, it can be easily detected if we ran out of address space by checking if the sum of the number of most significant bits used for spacing and the bit-length of the corset is greater than 64. This will help to diagnose the impact of our maximal spacing approach.

3. Since in a typical application the maximal expected alignment is no more than **`sizeof`**`(max_align_t)` and by extension 4KB for applications that use `mmap`,

the corset will likely be smaller than 14 bits. This gives the spacing about 50 bits to work with at all times, which seems like a sufficient amount even if the largest padding is initially wasted on the smallest allocations.

This might only become a problem when applications that use pointer tagging themselves need to be supported. This would suddenly decrease the amount of spacing bits to about 32 and padding would rapidly become scarce. The solution seems to lie in a form of spacing that takes into account the size of the allocation, such that padding is not equally distributed based on the difference between neighbouring base pointers but on the difference between the end of an allocated region and the start of the next one in unified space. We were not able to find a mapping with those properties that had any practically realistic overhead though.

Note that this alignment check mitigation is not strictly coupled to any of the other ASB mitigation strategies. It does not rely on any unified address space or registration of logical allocations. The base addresses of all logical allocations should simply fit the corset when they are allocated, however that may be implemented. The fact that the unified base addresses must also fit this corset is an artifact of an imperfect detection strategy, that might cause an alignment check on a unified value instead of a variant-specific value.

Finally, the most important consideration in this alignment mitigation is its influence on the diversification of the variants. The trap bits essentially provide hard constraints for pointer values, directly limiting memory layout diversification. Hence, an attacker knows the value of the

$$C = \left\lceil \log_2 \left(S + 2^N \right) \right\rceil$$

least significant bits for every pointer value, as they are the same in the variants. Previously, only the N least significant bits of every pointer value were known to the attacker. The only additional information provided by the corset restriction is that the offset pointers into the variants will all have the same alignment as well. It is currently unclear how this could aid an attacker during an exploit, as this only applies to the addresses within allocations, the layouts of which are not randomized by non-distributed MVEEs anyway.

4.3 Security impact of the proposed approaches

The detection approach with the best ASB mitigation presented in this work is the pointer-granular approach, as it ensures that the effect of any use of a pointer value that is accessible in the program is the same in all the variants. However, as previously discussed, it is also the worst from a security perspective since all pointer arithmetic happens in unified space and all memory errors that the programmer can make are trivially also ensured to have the same effect in all variants, directly counter-acting memory layout diversification. This enables attacks that the MVEE can normally protect against, such as a ROP-attack [10] that hijacks the control flow of the program by overwriting the value of a function pointer. As the function pointer exists in unified form by default, it is deunified when used in an indirect call operation. An attacker can therefore trivially locate the unified locations of arbitrary functions during the development of the attack by counting the occurrence index of that function beforehand, and, using for example a unified buffer overflow to write

that value to a function pointer, redirect all variants to the same attacker-controlled target.

Depending on the abstraction level at which the instrumentation of `load`/`store` operations is performed, not all MLD is counter-acted by this approach. An example of this are the return addresses that are pushed onto the stack before a function is called. This operation is not manually written by the programmer, and neither is the `load` operation that happens when the program jumps back to the location that the return address specifies. Hence, no deunification operation is applied when returning from a function and an attack still needs to smash the stack with variant-specific code pointers to hijack control flow, which the input replication and the disjoint code layouts in the variants mitigate [10].

Hence, a solution could be to simply not register or unify any code pointers. Although this prevents an attacker from successfully hijacking control flow in all variants, this does not prevent some of the data-only attacks that the MVEE could previously protect against. For example, a recently disclosed buffer overflow in the `sudo` program allowed an attacker to overwrite the string arguments to a dynamic symbol name lookup, the result of which was later the target of an indirect call [59]. If the effect of all buffer overflows is the same in all variants, this would still go undetected in GHUMVEE with our pointer-granular ASB mitigation applied. Conversely, if no ASB mitigation were applied, the buffer overflow would write to different locations in the variants' memory and the argument to the dynamic symbol name lookup function could not be reliably controlled by the attacker.

At the root of these problems lies the deunification operation that, by definition, produces a logically equivalent variant-specific value for every in-bounds unified value it is passed. To ensure that the memory layout diversification of the variants can not in any way be subverted, the deunification operation may only be applied to unified values that did not undergo modification after they were created, since we can not verify in the general case if any modification made them go out of bounds. In practice, if we can detect that a unified value does not undergo modification before it is deunified, we can simply not unify it in the first place. Hence, any practical use of the deunification operation is inherently insecure. However, without using the deunification operation, it becomes hard to mitigate some examples of ASB. This is highlighted by a slightly altered version of Listing 3.1, shown in Listing 4.7.

```
std::vector<int32_t> rel_ptrs;
char* base_ptr = (char*) malloc(sizeof(some_struct));
size_t N = 10;
for (int i = 0; i < N; i++)
    rel_ptrs.push_back((char*)malloc(sizeof(some_struct))
                                    - base_ptr);
std::sort(rel_ptrs.begin(), rel_ptrs.end());
std::vector<some_struct*> abs_ptrs;
for (auto rptr : rel_ptrs) {
    some_struct* abs_ptr = base_ptr + rptr + argc;
    printf("abs_ptr: %p \n", abs_ptr);
    *abs_ptr = { /* untrusted data */ };
}
```

Listing 4.7: An example address-sensitive use of relative pointers

If the relative pointers in `rel_ptrs` are unified, i.e. represent the difference between two unified pointers, the reconstruction of `abs_ptr` must first unify `base_ptr`, then add `rptr + argc` to it, and then deunify the result. Hence, this is not possible without deunification operation, but because `argc` might be attacker-controlled, a deunification operation would definitely be unsafe here. Alternatively, if the values in `rel_ptrs` are not unified to avoid the need for a deunification operation, the sort operation will cause the absolute pointers to be reconstructed in a different order, leading to a divergence when they are printed out.

However, the unification operation does not have to be applied at the moment that the relative pointers are constructed to mitigate their ASB. At the end of Chapter 3, we suggested to instrument the comparison operations inside the `std::sort` operation instead, to return some variant-agnostic value that guarantees consistent ordering of the relative pointers in the variants. Detecting these is complicated by the fact that a sort operation does not necessarily incur any comparison operations, as previously shown in this chapter in Listing 3.5.

This suggests that it is impossible in the general case to automatically mitigate all ASB in a program without using a deunification operation. However, it must be possible to manually mitigate all ASB in an arbitrary program without deunification operations. If not, that would mean that having unsafe ASB is an essential part of the programs functionality, which cannot be achieved without it. In our research, we found no evidence that such programs exist.

As an alternative to removing the deunification operation, we provided a stochastic solution that still uses the deunification operation, but inserts as much padding as possible in between allocations in the unified address space, to maximally detect which logical allocation a unified value is out-of-bounds for, such that we can translate it back to an equivalent out-of-bounds pointer for the variant-specific allocations. This is based on the assumption that most out-of-bounds accesses will happen close around the allocation and excludes many invalid memory accesses in programs from being exploited, especially when

there are few entries in the occurrence list. Consider again the unsafe deunification of the reconstructed unified `abs_ptr` in Listing 4.7. If `argc` is a 32-bit unsigned integer, it can maximally cause `abs_ptr` to go 4GB out of bounds of its allocation. As discussed, when using the spaced out mapping with dynamic padding, this will only lead to an insecure deunification if there are $\sum_{n=1}^{31}(2^{n-1}) = 2^{31} - 1$, or roughly 2 billion logical allocations registered, which is unrealistic for many real-world programs. Hence, in some cases, the ASB mitigation does not prevent the MVEE from detecting a divergence.

The next chapter reports on the development of a prototype implementation of one of the approaches presented in this chapter.

Chapter 5

Implementation details

We implemented a subset of the work presented in this thesis and evaluated it on the GNU core utilities. Specifically, the operation-granular detection approach was selected because it provides a benchmark for most of the claims in Chapter 3 about how ASB is introduced into a program. The approach was not combined with automated analysis. For the mapping, the count-by-allocation system with full size information was implemented as it is the simplest mapping that supports all features necessary to be used in conjunction with either of the detection approaches. No spaced out iteration order was used. We did not integrate the solution into any MVEE, primarily because GHUMVEE did not support running the GNU core utilities' integrated test suite. Sometimes a workaround was therefore needed to provide functionality that would otherwise be provided by the MVEE.
The registration of logical allocations will be investigated first, after which the details of the mitigation strategy for uninitialized data are outlined. Then, the instrumentation rules specified by the operation-granular approach are implemented. Lastly, the implementation of the selected count-by-allocation mapping is covered.

5.1 Registering all logical allocations

To build the count-by-allocation mapping, all logical allocations need to be registered as soon as their address could be referenced. Section 3.4 presented an exhaustive list of 3 operations through which addresses enter: a call to a dynamic allocation routine[1], an address-of operation and the decay of an array or function into a pointer. As discussed at the beginning of Chapter 3, no assumptions can be made about the value of any of these addresses in the face of full MLD. A fully compliant solution would therefore need to detect these operations statically and wrap them with a call to a registration function. Although detecting dynamic allocation calls and array/function decays might not be problematic, the address of variables that refer to arbitrary locations inside a logical allocation can be taken [21, Section 6.5.3.2.1], complicating the registration process. Consider the snippet in Listing 5.1.

[1]Note that this was broadly interpreted in Section 3.4 to also include OS-level calls such as `mmap` and `(s)brk`, as well as the non-standard `alloca`, which allocates on the stack.

```
std::unordered_set<int*> ints;
struct {
    int previous_field;
    int some_field;
} __attribute__((packed)) example = {};
int* second_field = &example.some_field;
ints.insert(second_field);
ints.insert(second_field - 1);
```

Listing 5.1: An example of an address-of operation that points to the middle of a logical allocation.

In this contrived example, the registration will first be called on the second field, after which a pointer is derived from its address to reference the first field and used address-sensitively. The second pointer will be unified but no entry in the occurrence list will contain it. The obvious solution to this problem is to register stack allocations instead of address-of operations. However, GHUMVEE, which this implementation targets, does not diversify the internal stack layout [4]. We can therefore regard the entire stack as a single logical allocation in this implementation and not instrument the address-of and array/function-to-pointer decay operations. Although this violates the initial premise that the ASB mitigation cannot rely on MLD-specific details, we do not consider this a shortcoming of the theoretical approach, but rather a shortcut taken to reduce implementation complexity.

This excludes global variables from registration, so we need additional instrumentation to support those again as well. In essence, we divide the address space up into three regions: global data, stack data and dynamically allocated data. We cover the registration of addresses from each region separately.

Registering global data

Global data should all be registered before the entry to `main`, since it can be used address-sensitively from that point on. We took a similar approach as CaVer [60], where the authors used an LLVM instrumentation pass to insert constructor functions into every translation unit that invoke a registration call for all global data declared in that translation unit. Constructor functions are a non-standard GNU extension of both C and C++ that are guaranteed to run before `main` is called [61]. Although the order in which they are invoked can be specified within a translation unit, it is unspecified across translation units. It was consistent on every program run in our testing though, which is important since the registration order of globals directly determines the unified addresses they map to, which should be the same across variants.

Concretely, we developed an optimization pass for LLVM IR that inserts an `init` function and adds it to the global constructor list of the module [41]. All of the functions in this list have a numerical priority associated with them, and they are executed in ascending order of priority. Ideally, the function that registers the globals is ran before any other constructor function in this list as all of them could contain ASB for which unification/deunification calls are inserted, but we disregarded this in our implementation and it did not cause any

issues.
Secondly, we iterate over all global variables that are accessible to the program[2] and insert calls to the `register_global` function that inserts them into the occurrence list. Since the count-by-allocation mapping requires size information, that is also deduced from their type. LLVM IR additionally contains alignment information about all variables, which is also passed to the registration function. There is no accompanying `unregister_global` function, as we assume they remain accessible throughout the duration of the program. This is again a simplification, as the global data in dynamically loaded libraries could be made inaccessible before the end of the program using `dlclose` [62], and subsequent loaded libraries could reuse the same memory regions for their global data, leading to the registration of an already registered region. However, this did not occur in our testing.

Note that registering all globals is an over-approximation to make sure that all globals whose addresses are accessible in the program are registered. Although this increases the memory overhead and hurts the run-time performance of every subsequent operation on the occurrence list, performance is not the main evaluation criterion for this implementation.

Registering stack data

As discussed in the introduction to this chapter, we consider the stack region to be a single logical allocation. The memory region occupied by the stack can be read from the `/proc/self/maps` file on Linux [63]. However, during our implementation we noticed that the offset of local variables to the end of the stack region[3] was not consistent on every program run. Instead, the offset shifted with multiple hundreds of bytes every time. We speculate that this was caused by the interaction between ASLR and the alignment of environment variables at the base of the stack, but we did not investigate this further. Instead, we took the address of a local variable inside the function which allocates the occurrence list and runs before any of the global registration calls happen (see later) and assumed that, plus some fixed offset, the obtained address would approximate the stack base. A stack size of 8MB was assumed with no support for enlargement. Note that these approximations are not inherent to the approach: if the solution were integrated into GHUMVEE, we could read a more precise stack region for every variant.

By registering the whole stack region this way, we do not need to explicitly support calls to `alloca`, as it will merely expand some stack frame that is already registered.

Registering dynamically allocated data

To register dynamically allocated data, a number of different solutions were available, like using link-time symbol wrapping [64] to resolve references to our interposers instead of the real functions, but that would not catch the internal `malloc` calls in libc functions like `strdup` and `readline`, which also allocate memory. Another solution would be to modify *glibc* and insert the necessary registration calls manually into the allocation functions, but we wanted the registration to easily work across different `malloc`

[2] Some of the globals, specifically the variables starting with "`llvm.`" are not accessible by the target program and are just used to hold meta-information about the module by LLVM.

[3] We assumed a stack with a downwards growth direction, so the end of the stack memory region corresponds to the logical start of the stack.

implementations as full MLD supports those. In the end we interposed the necessary functions by linking the application to a shared library that contained implementations for the functions before linking to libc. This resolved all calls to malloc to our wrapper functions. An equivalent alternative would be the use of the `LD_PRELOAD` environment variable, but that would have complicated running the integrated test suite of the GNU core utilities. The functions to wrap are the following: C11 specifies `malloc`, `calloc`, `realloc` and `aligned_alloc` [21, Section 7.22.3]. GNU's *glibc* additionally supports `reallocarray`, `pvalloc`, `valloc`, `posix_memalign` and `memalign` [65, 66], which do not call `malloc` but an internal allocation function instead.

The system calls `brk` and `sbrk` should be wrapped as well for full functionality. However, the default *glibc* `malloc` implementation uses `(s)brk` for allocations in the main arena [67], so it can not be used safely by the application and should not be called. We therefore omitted wrapping these.

Lastly, the `mmap`-family of system calls is somewhat unsafely supported by interposing their wrapper functions in the libc API. Of course, an application could manually invoke the system call without using the wrapper function, but this is unlikely. This is also an approximation since the solution is not integrated into GHUMVEE, which already intercepts all `mmap` calls since they are system calls. In total, we wrap calls to `mmap`, `mmap2`, `mmap64`, `munmap` and `mremap`.

We do not call the original functions using `dlsym` with `RTLD_NEXT` as would be typical. This was complicated due to the fact that `dlsym` itself calls `calloc` during its operation. A full solution would be to use a hand-written temporary allocator just to cover memory allocations during `dlsym`, but as a workaround we just called the respective internal symbols that *glibc* uses for these functions instead, which are prefixed by `__libc_`. Note that this restricts us to only use *glibc*'s `ptmalloc` in this implementation.

The dynamic allocation calls can be loosely subdivided into three groups: those that allocate, those that reallocate and those that free. Most of the allocating functions were implemented on top of each other instead of calling into their respective counter-part in *glibc*. As such `valloc`, `pvalloc`, `aligned_alloc` and `posix_memalign` were all implemented on top of `memalign` given that they have only slightly different behaviour. Similarly, `malloc` was implemented to call `calloc`[4], and `mmap64` and `mmap2` call `mmap`. The high-level operation of `memalign`, `calloc` and `mmap` is alike: call the real implementation, then register the result. Additionally, we took care to preserve the value of `errno` after the call to the real implementation and restore it on function exit to ensure maximal transparency of the registration process to the program.

The `mmap` wrapper is a bit more involved than the `malloc`/`memalign` wrappers, as it specifies arguments like `MAP_FIXED` and `MAP_FIXED_NOREPLACE` that try to map pages at a fixed address [22]. If a mapping already exists, it is replaced by the new mapping, unless `MAP_FIXED_NOREPLACE` is specified. The call fails if the programmer attempts to map pages to addresses that are not valid for its process. Because `malloc` also maps pages, the use of `MAP_FIXED` is rather dangerous as it might invalidate an arbitrary amount of pointers in the program. If it unmaps pages used by `malloc`, that is a bug that will eventually lead to a divergence, so we can just register the new area and move

[4] This was because as part of the uninitialized data mitigation, we need to zero out the result of `malloc` anyway.

on. If it does not and it fails in some variants, it should fail in all variants with EINVAL. If it does not fail in any variants and does not replace any malloc pages, it is unclear whether it should be allowed to replace existing mappings only in some variants. For MAP_FIXED_NOREPLACE, the program might use this to test if it is possible to extend a mapping without automatically unmapping the pages that is tested for, similar to mremap. The result of this can therefore differ in the variants, as some will not contain mappings at the specified address but others might. To have consistent results in the variants, the result of the call should therefore be checked across all of them before it can be decided to support the new mapping. This is not possible outside of the MVEE however, so we just always return MAP_FAILED with an EEXIST error code instead. Note that for MAP_FIXED, this changes the semantics as it can normally only fail with EINVAL. This did not cause issues in our testing however.

The reallocating functions are designed to always move the allocation somewhere else, unless in some cases when the new size is smaller than the original size. As such, realloc simply allocates a new block using malloc and copies the allocation contents there. Since the mapping stores the size of the allocation, that can trivially be retrieved. Most malloc implementations also support malloc_usable_size, which returns the size of the allocation that was actually used inside the implementation. This can be used in conjunction with mappings that do not keep full size information. A more efficient implementation could first test whether the new size is smaller than the old size and then check if the actual realloc call moves the mapping. If not, no re-registration needs to happen.

For mremap [68] the implementation was a bit more involved as it may be used to replace mappings or test whether an adjacent page is mapped. All of the mremap functionality is supported, except growing the mapping in-place as that might have different results in the variants. When called with a new size that is bigger than the old size, the default behaviour of mremap is to try and grow the mapping in-place and return an error if it fails. It can be made to move the mapping elsewhere if it can't grow in-place by specifying the MREMAP_MAYMOVE flag. In either case, if the new size is bigger than the old size, we first try to mmap a page adjacent to the region specified by the old size using MAP_FIXED_NOREPLACE. This ensures that the mapping cannot grow in place, after which the actual mremap call is executed. If MREMAP_FIXED was specified and the call succeeded, we unregister any previous registrations of the new region, as mremap will replace existing mappings if MREMAP_FIXED is specified. We then register the new region again, and, if MREMAP_DONTUNMAP was not specified, unregister the old region.
If the new size is smaller than the old size, we simply unregister the difference and forward to the real mremap.

Lastly, the functions that deallocate are straightforwardly implemented by calling the real deallocation function and also unregistering the associated region from the occurrence list. For munmap, this currently causes a small semantic change as the unregistration function will quit the program if the specified range was not previously registered. This is a measure to detect invalid deallocation calls. However, it is explicitly allowed to call munmap on pages that are not currently mapped [69]. Again, this caused no issues in our testing.

5.2 Zeroing out all logical allocations

In Chapter 4 we suggested to zero out logical allocations when they are first allocated to rid them of any uninitialized and variant-specific data they may contain. Chapter 3 shows that this is only necessary for stack and heap allocations, as global data is either zero or initialized at program startup and memory obtained directly from the operating system through `mmap` is already zeroed for security reasons. The `(s)brk` functions could technically return uninitialized data when used in a certain way, but since they are not supposed to be used directly by the program but rather only by the `malloc` implementation, zeroing out heap allocations will cover that as well.

Zeroing out heap allocations

Given that we interpose all heap allocation functions already, we can trivially zero out the allocations before returning the pointer to the newly allocated region. In the case of `malloc` this is done by directing the call to `calloc` instead, which zeroes out allocations by definition [21, Section 7.22.3.2]. For `memalign` a call to `memset` is inserted before returning. All other heap allocation functions are implemented using either of these two, so are therefore zeroed out as well.

Note that, depending on its implementation, `realloc` might require special care to make sure its data is zeroed. In our implementation, it always calls `malloc` and copies the allocation contents into the newly allocated block. Even when the new block is bigger, the extra bytes will still be zero because our `malloc` zeroes memory. However, if the call to `realloc` is instead allowed to go through to the actual `realloc` implementation of the allocation algorithm, it might not always allocate a new block to grow the allocation, but coalesce some adjacent free block into the allocation instead. Even if `free` were to zero memory[5], there might still be allocation-specific metadata contained between the end of the new region and the end of the old region. Alternatively, even if growing the allocation does allocate a new block, that call might not necessarily go through one of the functions we interpose but use some internal function instead. Hence, to zero out the bytes gained by a growing reallocation, a `memset` should be inserted that zeroes out the bytes between the end of the old block and the end of the new block. Since we implemented our own `realloc` instead of calling into the actual implementation, this was not necessary for us.

Also note that typically, when interposing libc functions, care needs to be taken to make sure it is compatible with the semantics the compiler assumes for them [70, 71, 46]. For example, based on the knowledge that `malloc` has no side-effects, the compiler could optimize away redundant calls to `malloc` and `free` [72], or potentially assume that memory read from `malloc` is uninitialized at first, enabling a host of additional optimizations [46]. However, it does not matter for our work if a redundant allocation call is optimized away or if the compiler assumes data is uninitialized while it is not anymore: we are not trying to remove the bugs that are in the program due to reading uninitialized data, but rather making sure the effects are not different in the variants. Hence, the semantics we provide for these functions are more strict than the ones the compiler assumes. This makes it unnecessary to disable some optimizations, as would typically be done for an interposed

[5] To clarify, `free` does not zero memory in our implementation, nor is it required.

allocation library using the `no-builtin-...` flags [70, 71].

Zeroing out stack allocations

To zero out stack allocations, calls to `memset` could be inserted after every local variable declaration. However, previous research has shown that the run-time overhead of doing that can be large [72]. Modern compilers have recently gained out-of-the-box support for flags that enable the automatic initialization of stack variables, including struct padding, with either zero or a predefined pattern [36, 73, 74], mostly out of security concerns to combat information leaks [74]. We therefore simply used Clang's `-ftrivial-auto-var-init=zero` flag. Clang developers have expressed concern that the ubiquitous use of this flag may cause programmers to rely on zero-initialization for future programs however [36], and are pushing the use of pattern-initialization as a solution. The long-term goal is to remove support for automatic zero-initialization from the compiler entirely as soon as the observed performance regressions for pattern initialization are mediated, made clear by the mandatory use of a long flag in conjunction with automatic zero initialization. For our purposes it is unimportant whether zero-initialization or pattern-initialization is used: as long as the data is initialized to the same value in all variants, it will not cause divergences.

5.3 Instrumenting the program with unification calls

As discussed at the beginning of this chapter, the operation-granular approach is taken to ensure that no ASB can occur in the program. We showed in Chapter 4 how this approach comes in two flavours, depending on the support for relative pointers. In this implementation, we chose not to support those as we could not find any use of them in the GNU core utilities. Hence, the only instrumentation rule we applied was to unify the conversion from pointers to integers and deunify the conversion back.

LLVM provides dedicated instructions for converting pointers to integers and back [41]. We therefore implemented the instrumentation by extending the LLVM IR optimization pass that registers the global variables to also iterate the instructions in the module and replace all `ptrtoint` instructions with calls to `unify`, and `inttoptr` instructions with calls to `deunify`.

This over-estimates the amount of pointer-to-int conversions, including implicit ones, that the program actually contains, as shown by Lee et al. [75, 76]. In their experiments, 86% of `ptrtoint` instructions were generated by LLVM itself, not by the programmer [76]. In an attempt to reduce overhead because of this, we modified the Clang front-end to tag all conversions between pointer and integers with an `isExplicit` attribute that signified that the conversion was emitted as part of the initial IR generation process and represented a real conversion expressed in the source code. This allowed us to only replace those conversions with `unify`/`deunify` calls instead of all of them. This works well for the majority of ways in which pointers can be converted to integers. In fact, for all of the possible conversion methods discussed in Section 3.4, Clang successfully detects them as type conversions for their most simple forms and, even without optimizations, replaces them with the appropriate instructions. However, this falls apart

when slightly more complex constructs are used. For example, in the `obstacks` implementation of *Gnulib* [43], the `__PTR_ALIGN` macro essentially expands to a construct[6] like `NULL + (ptr - (false ? other_ptr : NULL))` for which the Clang front-end does not emit a `ptrtoint` on `ptr`, even though the programmer fully intended this. However, it does emit an `inttoptr` on the addition of an integer to the `NULL` pointer, causing a non-unified pointer value to be deunified and crashing the mapping. This exemplifies why it is not sufficient to only instrument conversions that the front-end could detect. When the program is compiled with optimizations instead, the optimizer folds the conditional evaluation of `false` into `NULL` and a `ptrtoint` is emitted. In addition, some of the tagged instructions also disappear due to the optimization passes that run on the IR. To resolve this, we instrumented all `ptrtoint`/`inttoptr` instructions instead, and optimizations were turned on. This exposes our solution to the many conversions that the compiler generates on itself, increasing the overhead of our implementation. Additionally, just like Lee et al. report in [75], some optimization passes emit instructions that essentially type pun pointers into integers and back, without `ptrtoint` or `inttoptr` instructions. Listing 5.2 shows a real snippet of generated IR at optimization level 3 from `cat.c` in the GNU core utilities.

```
%arrayidx = getelementptr inbounds i8*, i8** %argv, i64 %some_idx
%1 = bitcast i8** %arrayidx to i64*
%2 = load i64, i64* %1, align 8
store i64 %2, i64* bitcast (i8** @infile to i64*), align 8
%3 = inttoptr i64 %2 to i8*
```

Listing 5.2: A piece of LLVM IR generated at optimization level 3 that uses type punning to convert a pointer to an integer.

Listing 5.3 shows the equivalent C pseudo-code[7]. A pointer to one of the strings passed in `argv` is obtained, called `arrayidx`. Then, the pointer value `arrayidx` refers to is converted to an integer using type punning. This should have incurred a `ptrtoint` instruction. The pointer value `infile` is then set to this integer value, essentially equating it to `*arrayidx`. This is again done using type punning, by reinterpreting the address of `infile` to be a pointer to an integer instead. This operation should technically have incurred a `inttoptr` instruction, but since the integer that is converted from was obtained using type punning and is therefore not unified, it actually prevents the deunification of a non-unified value in this case. Finally, the integer is cast to a pointer the regular way, causing a deunification on a non-unified value anyway.

[6] For brevity, the casts necessary to make this construct compile are omitted.

[7] Note that this is not the code that produced this IR, but merely a C representation to show the type punning construct more clearly.

```
char* infile;
char** arrayidx = argv + some_idx;
// type pun '*array_idx' as integer
i64 var2 = *((i64*) arrayidx);
// type pun 'infile' as integer
*((i64*)&infile) = var2;
char* var3 = (char*) var2;
```

Listing 5.3: C pseudo-code representing Listing 5.2.

These compiler-generated type puns are currently not detected in our implementation and crash the mapping. A more robust approach would be to follow Lee et al.'s method of removing these, by disabling the specific optimizations that cause this [75].

5.4 Implementing the unified address space

We used the count-by-allocation mapping with full size information to provide unified address values to the variants, mainly because mappings can easily be padded with dummy allocations to reach suitably aligned unified addresses. Similarly, `mmap` is naturally supported by mapping all pages in a region to consecutive unified addresses, such that they can individually be unmapped. The main disadvantages of this mapping are its poor run-time performance with linear unification and deunification complexity, 16-byte per-allocation memory overhead and, depending on their placement, very tightly fit logical allocations. However, neither performance overhead nor security implications are the main metrics that this implementation will be evaluated on.

We implemented the mapping in C++17 using two STL vectors; one to store the base pointer and size entries for every logical allocation and one to store whether the logical allocation in the first list at the same index is currently free or not. We refer to the former as the *alloc list*, and the latter as the *free list*. We did not store the booleans in the free list in the alloc list entries to avoid increasing the memory overhead even further. Instead, we used a `std::vector<bool>` for the free list which is may store every boolean as a single bit [31, Section 26.3.12], and on most implementations it will. We still use the term *occurrence list* to refer to the abstract list of metadata entries that make up the mapping, regardless of how that is implemented.

The basic registration procedure takes a base pointer, a size S and an alignment boundary B and uses a simple first-fit allocation algorithm to find a free entry in the occurrence list that contains the range $\left[K * 2^B, K * 2^B + S\right[$ with K some non-zero positive integer ($K \in \mathbb{N}_{>0}$). If no entry is found, it allocates an extra entry at the end of the occurrence list. If there is spacing between the start of the free region the allocation fits into and the unified value at which it should be mapped[8], an additional free entry is inserted to pad the

[8] Note that the value of $K * 2^B$ does not fully satisfy the requirements for unified base pointer values as specified by the alignment check mitigation in Chapter 4. Specifically, it accounts for neither the trap bits nor the fact that it might accidentally be aligned to a bigger boundary than the variant-specific base pointer values. However, this did not cause issues in our testing.

entry until its unified value is $K * 2^B$. Note that it is not technically necessary that the registration functions receive any alignment information about the base pointer values. If the solution were integrated into GHUMVEE, we could find the highest boundary that the pointers in all variants support and assume that as the alignment boundary of the allocation. However, that might accidentally over-estimate the real alignment of the allocation and lead to an unnecessarily large corset on the base pointer values, likely wasting memory in the variants and address space in the unified address space.

A deregistration operation finds the entry in the occurrence list this pointer belongs to, updates the free list to signify its deallocated status and coalesces adjacent free blocks to combat fragmentation.

For `mmap`'ed pages, a more high-level registration function is provided that calls the aforementioned registration function once per page while making sure that the pages are also consecutive in the unified address space.

The unification and deunification procedures are implemented exactly as described in Chapter 4. They are both $O(n/2)$ for in-bounds pointers and worst-case $O(n)$ on out of bounds pointers.

Due to the way we register global variables, the registration function is likely called before the global variables in the translation unit that contains the occurrence list are constructed. Therefore, the alloc and free list are stored as members of a `metadata` class, of which the only instance is a local static variable in a function that returns a reference to that variable. All accesses to the metadata must therefore call that function. As local static variables are constructed when the function they belong to is called for the first time [31, Section 9.7.4], it does not matter which order the translation units are initialized in: the metadata instance and the corresponding vectors will always be constructed first.

During the registration and unregistration calls, the occurrence list itself might grow or shrink and therefore invoke dynamic allocation routines which are interposed to call the registration function, potentially leading to an infinite recursion and stack overflow. To avoid this, we introduced a global *hook* flag, that every interposer function queries to determine if it should call the registration functions or not. Inside all the functions that operate on the occurrence list we clear this flag for the duration of the function, which guarantees that interposer functions will pass their calls through to libc without registering anything. In fact, a generalization of this hooking system is necessary to support different allocation strategies: the monitor must not terminate execution if different allocation algorithms call different system calls at different times.

Chapter 6

Evaluation

To evaluate our implementation described in Chapter 5, we created a program that implement all of the ASB cases from Chapter 3. In this chapter, we will discuss the results of our evaluation of the performance, completeness and transparency of our implemented ASB mitigation. The precise ASB tests that we ran are listed below:

1. Hash table pointer insertion and iteration.
2. Alignment checks, strictly enforced by system calls that require page alignment.
3. Relative pointers across allocations.
4. Writing out padding bytes and uninitialized variables.
5. Writing out pointer variables, both as integers and as pointers.
6. Sorting an array based on pointer variables.
7. Storing non-address values in pointer variables.

The final case is not ASB but was included to highlight how other unrelated coding patterns can interact with the ASB mitigation. We implemented a dynamic array and simple hash table ourselves to support the testing, as GHUMVEE loads its own patched standard C and C++ libraries, which we did not instrument.

Evaluation of performance impact

We compiled these cases using Clang 11.0.0 with optimization level 2 and accumulated their execution time 100 times on a 4-core, 2.8 GHz[1] Intel Core i7 Ubuntu Linux 20.10 machine with 16 GB of RAM to obtain the results reported in Table 6.1.

[1] We disabled the turbo on the CPU to get consistent results between runs.

	Native	**With registration**	**`isExplicit`**	**Full instrumentation**
Full	130ms (100%)	15.5s (11823%)	15.5s (11823%)	crash on `rel_ptrs`
No `rel_ptrs`	90ms (100%)	12.8s (14122%)	12.8s (14122%)	14.9s (16456%)

Table 6.1: Performance measurements of the prototype implementation.

From this data, the main performance culprit seems to be the registration and zeroing of logical allocations. To validate these results, we obtained micro-benchmarks for the registration code specifically. On a sequence of 10000 allocations randomly varying in size from 1 to 1000 bytes, the overhead of using `calloc` instead of `malloc` turned out to be already around 100%. We observed a slowdown of factor 950 compared to the base performance of `calloc` when additionally registering its allocations in the occurrence list. Dropping the alignment requirements for the unified addresses reduces this to a factor 450. This makes sense based on the way unified alignment is implemented, as discussed in Chapter 5: up to 3 entries are inserted into the occurrence list for every aligned allocation request.

However, this drastic performance hit is not an inherent property of registering allocations. Contrary to the implemented count-by-allocation mapping which has to find free entries by iterating a list, the spaced out mapping with dynamic padding can just walk a series of predefined values[2] to generate new spaced out unified addresses in $O(1)$, as discussed in Chapter 4. A unification operation then reduces to an insertion at the end of a list. To overview this best-case scenario, we altered the micro-benchmark to push the returned pointer value into a dynamic array. This reduced the performance overhead of zeroing and registering allocations compared to raw `malloc` to under 10%.

A second observation of the data in Table 6.1 is that the instrumentation of explicit type conversions does not incur any additional overhead. This is a rather misleading measurement, as simply no conversions are being instrumented. As discussed in Chapter 5, the conversions tagged `isExplicit` by the front-end do not necessarily survive the optimizations and in this case, none of them did.

Finally, after instrumenting all `inttoptr`/`ptrtoint` instructions, the program crashed on the relative pointers case because it tries to store the difference between two pointer values back into a pointer variable. This is a combination of the relative pointers case and the case where non-address values are stored in a pointer. This conversion will incur a deunification operation, which, when it can not find a logical allocation that contains the value, will crash the program.
Alternatively, it could just assume this integer is not address-dependent, cast it to a pointer and return that instead. However, a problem arises when that pointer is cast back to an integer at a later point: the unification operation in some variants might find a logical allocation in variant-specific space that contains it. There is no way of checking for this

[2]Note that this is a memory-inefficient method that may not be suitable for every application.

at the moment of deunification either, because the content of the occurrence list could change arbitrarily before the unification operation.

Another option could be to either tag the returned pointer or cache its value so it can be detected at the unification operation that this was not actually a variant-specific pointer value. However, arbitrary arithmetic in the full 2^{64} number range might be performed on the value while it is a pointer, obscuring any trace of the value it had when it was originally deunified.

The only way it can be detected at unification time that this value is not a variant-specific value is because it has the same value in all variants. Therefore, we propose to check on every unification operation whether the pointer value to unify is the same in all variants, and if so, bypass all mapping and return its value instead. This also trivially handles pointer values like `NULL` and `MAP_FAILED`, although for performance reasons it might be desirable to explicitly check for those.

There still exists an edge case where a bug is introduced: the deunification operation can not tell from an integer if it is a unified pointer value or some non-address value. If it is a valid unified address that points into a logical allocation, it will be deunified to different but equivalent variant-specific values. If it then undergoes pointer arithmetic and goes out of bounds of the logical allocation, the unification operation will be on different variant-specific values and will not detect that this was originally an non-address value, which will make it return a different integer value than the programmer expects.

Ultimately, these issues are resolved if the detection approach does not depend on the language's type system, since a pointer that contains a non-address value will never be dereferenced by the program. This is only the case for the pointer-granular approach and its associated extension with automated program analysis.

When removing the relative pointers case, the program runs fine when all conversions are instrumented, although an additional overhead of over 2300% is observed. In all test cases together, there are no `intttoptr` instructions, and only 3 `ptrtoint`'s. Still, the unification operation is executed a total of 779 times in a single pass of the test cases. This is a worst-case scenario, as the program is specifically constructed to contain as much ASB as possible.
Additionally, no automatic program analysis was employed to reduce the use of these operations and due to extra free entries the alignment registration inserts, the unification operation has to walk almost twice as many entries as in the spaced out mapping with dynamic padding. By removing the alignment requirement, the overhead reduces to 2077% more than the non-instrumented registration with an alignment requirement of 1.

Evaluation of ASB coverage

The high-level structure of the ASB in the test program is as follows. First, we create a setup where dynamic and static[3] **`struct`** allocations are stored in a dynamic array. Then, we sort the array on ascending pointer values to make sure that the ordering is address-dependent. We then iterate over the array and pass every **`struct`** to the main test

[3]Due to an unresolved bug in the registration of stack allocations, there were not included in the testing.

function that contains the bulk of the ASB. The contents of all the **struct** members, as well as their addresses, are printed out. Out of all their members, only two are initialized: a pointer and an integer that both hold the **struct**'s own address. After that, it is verified for the `mmap`'ed allocations that they are aligned on a 4KB boundary and we check if any of the `malloc`'ed ones happen to be aligned on a page boundary as well. If that is the case, we make a dummy call to `madvise` whose first argument is required to be page-aligned [57] to enforce that the alignment check can not be a bogus value. Finally, to conclude the main test procedure, we write the content of the **struct** directly to `stdout` as a buffer using `write`, to cover the padding and the pointer-in-buffer case.

After this, we insert all the **struct**s into a hash table and repeat the iteration: the same function is called for each struct.

Then, we create a new dynamic array that contains relative pointers to all the struct based on the address of some global variable. As previously discussed, the integers are stored inside pointer variables. We then sort that array and call the main ASB function again on the reconstructed absolute pointers. This concludes the test program.

The implemented approach is not designed to mitigate all ASB of the test cases. It covers the hash table case, the sorted pointers array case, the integers that hold pointer values and all uninitialized variable cases. With some manual instrumentation help, also printing out pointer values can be supported.

We confirmed this for the aforementioned cases by running our test suite in GHUMVEE on an Ubuntu Linux 18.04 QEMU virtual machine. We disabled all test cases we don't support first to verify that our `asb` mitigation works. GHUMVEE with two variants detects a divergence in the buffer contents of the `write` system call when the struct contents are written to `stdout`. When recompiling the same program with our instrumentation enabled, all of the allocations are registered and zeroed out. When their value is being written to their integer member, our instrumentation unifies the value. When sorting the array, the pointer values are compared. LLVM IR emits `ptrtoint` instructions to do the comparison on the integer representation of the pointer values. Therefore, the comparison operates on the unified values instead and the order of the pointer values is the same in both variants. When the main ASB function prints out the members, they are either zero or a unified address, since the pointer member was disabled. The alignment check on `malloc` was disabled, because some unified addresses might incidentally align to a 4KB boundary and over-state the alignment boundaries of the variants' values. Finally, because the padding is also zero-initialized, writing out the **struct** as a buffer does not cause any divergences.

The insertion of pointers into the hash table incurs a pointer to int conversion. We did this using pointer arithmetic, by subtracting the `NULL` pointer from the inserted pointer. The front-end successfully detects this as a pointer to integer conversion and the unification operation is applied. Therefore, the subsequent iteration causes no issues.

We observe no divergences when the instrumentation is applied, whereas the monitor terminates the variants on every one of these cases without our instrumentation.

Evaluation of transparancy

To evaluate whether our solution is transparent to the functionality of the program, i.e. does not introduce bugs, we compiled both the GNU core utilities and *Gnulib* with our instrumentation passes applied and linked them with our interposer library. We then measured the amount of test cases in their integrated test suites that failed because of our instrumentation.

First, we enabled only the registration, to test that separately. Out of the 627 tests in the GNU core utilities' test suite, 517 passed and 110 were skipped. No tests failed. Without any registration, two more test cases are skipped instead of passing. This means that *with* our instrumentation, 2 more cases passed than *without* our instrumentation. Although this is not necessarily problematic, it highlights how the solution is not entirely transparent to the application. Since no observable error occurred in our registration system, we suspect this was caused by the use of GCC to compile the unregistered version. In the case of Gnulib, 315 out of 362 test cases passed with our registration disabled, 47 were skipped and no tests failed. When enabling our registration, 1 of the succeeding tests fails, namely a regex parses which gets stuck in an infinite loop. We were not able to determine the cause of this issue.

After also enabling instrumentation of the `isExplicit` `ptrtoint/inttoptr` instructions at optimization level 0, 4 test cases in the core utilities failed, all because of the `__PTR_ALIGN` macro in Gnulib's obstacks implementation. As previously discussed, this is because Clang doesn't fold the evaluation of a `false` conditional ternary operator into the **`else`** expression there without optimizations.

Chapter 7

Conclusions and future work

In this work, we evaluated the problem of address-sensitive behaviour causing benign divergences in Multi-Variant Execution Environments (MVEEs) and proposed solutions to automatically mitigate it.

We gave an overview of the problem in Chapter 3 with some real-world snippets and examples, explaining how they introduce ASB and when they are used in practice. The main insight from this part is that ASB is widespread and includes common operations like checking the alignment of memory addresses. We then constructed a theoretical frame of reference for ASB to provide different perspectives on the ways it is introduced into a program, and how it causes divergences. We argued that ASB caused by uninitialized data is merely a side-effect of memory layout diversification and can easily be mitigated with existing solutions. Hence, its analysis was not the main concern of this work.

We identified that ASB caused by pointer variables does not necessarily cause divergences, and introduced the concepts of *safe* and *unsafe* ASB accordingly. Secondly, we made the distinction between *strong* ASB that is caused by interpreting pointer values for their properties as a regular number, rather than as a reference to some allocation, and *weak* ASB in which a pointer is only used to refer to an allocation as a system call argument. Within *strong* ASB, we found that a non-address value is often derived from an address value as a result of an address-sensitive operation, e.g. the result of an alignment check. This non-address value is still variant-specific, but not always used to make an assumption about the pointer value. In the hash table case, there is no information that the calculated index provides about the original pointer value that was hashed. However, in the alignment check case, if the result says that the pointer is aligned to a certain boundary, the program expects that it can be used in an operation that strictly requires that alignment, such as an aligned vector instruction. We called this set of assumptions that the program makes about a pointer value based on the result of an address-sensitive operation the *feedback* of that operation. If no feedback exists, such as in the hash table case, the address-sensitive behaviour is said to be non-reversed. Otherwise, it is reversed and care should be taken when mitigating its effects that no assumptions are broken.

In Chapter 4, we outlined multiple mitigation strategies to automatically neutralize the effects of ASB in the variants. We first provided an informal proof argument that neither the

use of static nor dynamic analysis would be able to provide full ASB detection coverage. Alternatively, we presented two rulesets for instrumentation logic that provided approximate catch-all protection against ASB, based on the insights into its causes and effects gained in Chapter 3. The pointer-granular approach outperforms the operation-granular approach in terms of completeness, as the latter is unable to detect every conversion from pointers to other types. However, especially for safe programs, the former incurs a considerably higher run-time overhead due to the instrumented load/store operations. Both suffer from similar security flaws for out-of-bounds accesses where they end up potentially making all variants access the same logical allocation on an invalid memory access, depending on the mapping that they use.

We also showed how these approaches can be optionally extended using automated program analysis. The examination of a concrete example suggested that the security and performance improvements can be the largest when it is used in conjunction with the pointer-granular approach.

To provide consistent values in all variants for address-sensitive operations, we created a unified address space and mapped it to the virtual address space of every variant. We showed that the implementation of this mapping influences the characteristics of the detection strategies significantly: if logical allocations unify to addresses that are numerically close together, the security problem with the out-of-bounds access is amplified as most of them occur close to the allocation. The key insight to construct these mappings is that the allocation order of logical allocations is the same across variants. We analyzed two mappings in detail: one which tightly fits consecutive allocations next to each other in unified space, and another which divides the allocation up into equally-sized segments and stores those next to each other. We showed how the latter could outperform the former in every regard if the segment size is well-chosen.

In addition to the automated program analysis, we investigated the use of more spaced out unified addresses as a way of minimizing the chances of hiding true divergences caused by memory errors. Notably, the breadth-first traversal of a complete binary tree of all 64-bit numbers was suggested to dynamically place new allocations as far away from previous ones as possible.

We also provided a solution to the remaining issue of alignment checks, based on the concept of a *corset* that all base pointers are forced into. This ensures that all offset pointers into every allocation, both in variant-specific space and in unified space, always support the the same set of alignment boundaries. We investigated the interaction of this additional constraint on the unified address space with the previously discussed spacing of the addresses and concluded that both have opposing interests when deciding the spacing for new allocations.

In Chapter 5, we presented a limited implementation of the operation-granular approach using a modified Clang compiler and LLVM IR instrumentation pass, supported by the count-by-allocation mapping. We evaluated its effectiveness, performance and transparency on a suite of ASB test cases in Chapter 6 and confirmed that it was able to

automatically mitigate all ASB, except in cases where our simplistic instrumentation based on the `ptrtoint/inttoptr` instructions did not suffice to detect all necessary conversions. Specifically, our implementation failed to automatically mitigate the use of sorted relative pointers, undetected pointers in buffers passed to system calls, and pointers that are formatted into strings and then passed to system calls. The main limitations of the implementation were the high performance overhead of the registration and unification procedures and its strong dependence on the specific instructions that Clang/LLVM used to represent syntactical constructs.

Future work

Although this work brings a lot of previously non-existent background to the issue of ASB in MVEEs, the main problem that remains unsolved is that of securely providing address-sensitive operations with the same values in the variants without neutralising MLD or introducing bugs. The security analysis at the end of Chapter 4 shows that even with the spaced out mappings we present, full ASB mitigation in the variants can not prevent that the ROP-immunity of GHUMVEE with disjoint code layouts is at least partly compromised. We suspect that this trade-off between ASB coverage and security is inherent to the applied memory layout diversification, and that future work will need to strike a balance between both.

The main benefit of MVEEs is the comprehensive protection against zero-day binary exploits they provide at a relatively low performance overhead. Although ASB limits their applicability in practice, a decrease in their security guarantees decreases their relevance as a secure protection mechanism entirely. Additionally, in a future production system, we expect programs to be ran inside the MVEE for a long time, e.g. a web server. Neither ASB, nor a weakened protection against ROP-attacks is tolerable in this scenario, but having the MVEE not terminate execution because of ASB is of limited relevance if it makes real attacks go unnoticed as well. Therefore, we propose that future research efforts focus on developing automated program analyses that can maximally assist a developer in neutralising ASB, e.g. by using a conservative static analysis that proves for as much of the code as possible that it does not contain unsafe ASB, and already instruments the code with unification operations if it can detect non-reversed ASB. For cases where it cannot make strong guarantees, it should not attempt to provide an approximation. Introducing a bug into the program that may or may not cause a divergence is objectively worse than having false positives because of ASB.

If future work accepts the premise that the deunification operation may not be used in the ASB mitigation, we propose that the mappings presented in Chapter 4 are revisited as they can likely be simplified. For instance, it may not be necessary anymore to register every allocation, or any at all.

However, future work could also continue investigating ways of safely providing deunification support, as we believe that fully automatic ASB mitigation requires it. To that end, we suggest the use of fat pointers in the unified space to further increase the spacing beyond any arithmetic that could be applied by the program. For example, the program

could be instrumented to store unified values in 128-bit wide integers, and explicitly disallow the addition of values greater than 64-bit to this value. Every unified value can then exist in its own 64-bit wide *domain*, that can always be safely translated back to an offset pointer around the allocation it was derived from.

Alternatively, it could be investigated whether existing solutions that provide bounds-checking for pointer values, like low-fat pointers [77], can be adapted to provide bounds checking for unified values instead. The key idea here is that unified values may not appear as frequently in the program as regular pointers, and explicit bounds checks only for them could thus incur a significantly lower performance impact. Additionally, the layout of the unified address space can be designed at will to better fit an efficient bounds-checking scheme.

Bibliography

[1] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and efficient application monitoring and replication," 06 2016.

[2] S. Volckaert, "Advanced techniques for multi-variant execution," Ph.D. dissertation, Ghent University, 2015.

[3] W. Commons. (2008) The memory format of an ieee 754 double floating point value. [Online]. Available: https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

[4] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "Ghumvee: Efficient, effective, and flexible replication," in *Foundations and Practice of Security*, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, A. Miri, and N. Tawbi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 261–277.

[5] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "Kmvx: Detecting kernel information leaks with multi-variant execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 559–572. [Online]. Available: https://doi.org/10.1145/3297858.3304054

[6] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 431–442.

[7] B. Salamat, A. Gal, and M. Franz, "Reverse stack execution in a multi-variant execution environment," in *In Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[8] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: A user space multi-variant execution environment," 2008.

[9] T. Pax, "Pax address space layout randomization (aslr)," *http://pax.grsecurity.net/docs/aslr.txt*, 2003.

[10] S. Volckaert, B. Coppens, and B. De Sutter, "Cloning your gadgets: Complete rop attack immunity with multi-variant execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 4, pp. 437–450, 2016.

[11] H. Shacham, E. Buchanan, R. Roemer, and S. Savage, "Return-oriented programming: Exploits without code injection," *Black Hat USA Briefings (August 2008)*, 2008.

[12] "Clang: a c language family frontend for llvm." [Online]. Available: https://clang.llvm.org/

[13] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, 2000, pp. 119–129 vol.2.

[14] A. Jajoo, "A study on the morris worm," 05 2018.

[15] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106.

[16] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 New Security Paradigms Workshop*, ser. NSPW '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 7–16. [Online]. Available: https://doi.org/10.1145/1900546.1900550

[17] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1871–1885. [Online]. Available: https://doi.org/10.1145/3372297.3417289

[18] B. Cox and D. Evans, "N-variant systems: A secretless framework for security through diversity," in *15th USENIX Security Symposium (USENIX Security 06)*. Vancouver, B.C. Canada: USENIX Association, Jul. 2006. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/n-variant-systems-secretless-framework-security-through

[19] B. Salamat, "Multi-variant execution: Run-time defense against malicious code injection attacks," Ph.D. dissertation, USA, 2009, aAI3359500.

[20] D. Poetzsch-Heffter, T. Bourguenolle, S. Volckaert, P. Larsen, and M. Franz, "Treating address sensitive behaviors causing benign divergence," UC Irvine, Tech. Rep., September 2015.

[21] ISO, *ISO/IEC 9899:2011 Information technology – Programming languages – C Draft N1548*. Geneva, Switzerland: International Organization for Standardization, December 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[22] [Online]. Available: https://man7.org/linux/man-pages/man2/mmap.2.html

[23] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dmvx: Secure and efficient multi-variant execution in a distributed setting," 2020.

[24] A. Voulimeneas, D. Song, F. Parzefall, Y. Na, P. Larsen, M. Franz, and S. Volckaert, "DMON: A distributed heterogeneous n-variant system," *CoRR*, vol. abs/1903.03643, 2019. [Online]. Available: http://arxiv.org/abs/1903.03643

[25] X. Wang, S. Yeoh, R. Lyerly, P. Olivier, S.-H. Kim, and B. Ravindran, "A framework for software diversification with ISA heterogeneity," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 427–442. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/wang-xiaoguang

[26] Mar 2021. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

[27] "Gcc, the gnu compiler collection." [Online]. Available: https://gcc.gnu.org/

[28] Corob-Msft, "Microsoft c/c++ documentation." [Online]. Available: https://docs.microsoft.com/en-us/cpp/

[29] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 260–275. [Online]. Available: https://doi.org/10.1145/2517349.2522728

[30] Corob-Msft, "Based pointers (c++)." [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/based-pointers-cpp

[31] ISO, *ISO/IEC 14882:2017 Information technology – Programming languages – C++ Draft N4659*, 5th ed. Geneva, Switzerland: International Organization for Standardization, Dec. 2017. [Online]. Available: https://www.iso.org/standard/68564.html

[32] P. Biswas, A. D. Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer, "Venerable variadic vulnerabilities vanquished," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 186–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/biswas

[33] "Dsa-1571-1 openssl - predictable random number generator." [Online]. Available: https://www.debian.org/security/2008/dsa-1571

[34] "Valgrind." [Online]. Available: https://valgrind.org/

[35] "Cppcheck." [Online]. Available: http://cppcheck.sourceforge.net/

[36] "Automatic variable initialization." [Online]. Available: https://reviews.llvm.org/D54604

[37] "The gnu c library (glibc)." [Online]. Available: https://www.gnu.org/software/libc/

[38] *Executable and Linking Format (ELF) Specification*, Tool Interface Standard (TIS) Committee, 1995, v1.2. [Online]. Available: ftp://download.intel.com/design/perftool/tis/elf11g.zip

[39] The IEEE and X/Open Company Ltd, *System Interfaces and Headers Issue 4, Version 2 - IEEE Std 1003.1, 1990 Edition.* Apex Plaza, Forbury Road, Reading, Berkshire, RG1 1AX, United Kingdom: X/Open, 1994. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9695969499/toc.pdf

[40] [Online]. Available: https://man7.org/linux/man-pages/man3/alloca.3.html

[41] "Llvm language reference manual." [Online]. Available: https://llvm.org/docs/LangRef.html

[42] "Int36-c. converting a pointer to integer or integer to pointer." [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/INT36-C.+Converting+a+pointer+to+integer+or+integer+to+pointer

[43] "Gnulib - the gnu portability library." [Online]. Available: https://www.gnu.org/software/gnulib/

[44] R. Krishnaiyer, Jul 2013. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/data-alignment-to-assist-vectorization.html

[45] A. Technology, "Amd64 technology amd64 architecture programmer's manual volume 1: Application programming," sep 2003.

[46] A. Milburn, H. Bos, and C. Giuffrida, "Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," Feb. 2017.

[47] "Exp33-c. do not read uninitialized memory." [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory

[48] D. Schoepe, M. Balliu, F. Piessens, and A. Sabelfeld, "Let's face it: Faceted values for taint tracking," vol. 9878, Askoxylakis, Ioannis. Springer Verlag, 2016, pp. 561–580. [Online]. Available: $$Uhttps://doi.org/10.1007/978-3-319-45744-4

[49] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," 2013.

[50] A. T. Tran, "An empirical study of alias analysis techniques," 2018.

[51] V. Rivera and B. Meyer, "Autoalias: Automatic variable-precision alias analysis for object-oriented programs," *SN Computer Science*, vol. 1, no. 1, p. 12, Jul 2019. [Online]. Available: https://doi.org/10.1007/s42979-019-0012-1

[52] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992. [Online]. Available: https://doi.org/10.1145/161494.161501

[53] A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter, "X86-64 instruction usage among c/c++ applications," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 68–79. [Online]. Available: https://doi.org/10.1145/3319647.3325833

[54] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[55] "24.3. memory management." [Online]. Available: https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

[56] "mprotect(2) - linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man2/mprotect.2.html

[57] "madvise(2) - linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man2/madvise.2.html

[58] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[59] "CVE-2021-3156." Available from MITRE, CVE-ID CVE-2021-3156., January 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156

[60] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 81–96. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee

[61] "Function attributes - using the gnu compiler collection (gcc)." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Function-Attributes.html

[62] "dlclose(3) - linux man page." [Online]. Available: https://linux.die.net/man/3/dlclose

[63] "proc(5) - linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man5/proc.5.html

[64] "Using ld, the gnu linker - options." [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

[65] "Replacing malloc (the gnu c library)." [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Replacing-malloc.html

[66] "Summary of malloc (the gnu c library)." [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Summary-of-Malloc.html

[67] "Malloc internals - glibc wiki." [Online]. Available: https://sourceware.org/glibc/wiki/MallocInternals

[68] "mremap(2) - linux man page." [Online]. Available: https://man7.org/linux/man-pages/man2/mremap.2.html

[69] "munmap(2) - linux man page." [Online]. Available: https://linux.die.net/man/2/munmap

[70] "gcc(1) - linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man1/gcc.1.html

[71] "Clang 13 documentation." [Online]. Available: https://clang.llvm.org/docs/CommandGuide/clang.html

[72] X. Yang, S. M. Blackburn, D. Frampton, J. Sartor, and K. S. McKinley, "Why nothing matters: The impact of zeroing." ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/why-nothing-matters-the-impact-of-zeroing/

[73] "[rfc][patch for gcc12][version 1] add -ftrivial-auto-var-init and variable attribute "uninitialized" to gcc." [Online]. Available: https://gcc.gnu.org/pipermail/gcc-patches/2021-February/565515.html

[74] J. Bialek, "Solving uninitialized stack memory on windows," May 2020. [Online]. Available: https://msrc-blog.microsoft.com/2020/05/13/solving-uninitialized-stack-memory-on-windows/

[75] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in llvm," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276495

[76] ——, "Safely optimizing casts between pointers and integers," 2019, 2019 EuroLLVM Developers' Meeting. [Online]. Available: https://llvm.org/devmtg/2019-04/slides/SRC-Lee-Safely_optimizing_casts_between_pointers_and_integers.pdf

[77] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 132–142. [Online]. Available: https://doi.org/10.1145/2892208.2892212

[78] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

Appendix A

Error-free counting sort without control-flow dependence on the data

```
void sort(uint32_t* data, size_t len) {
    static std::vector<bool> cnts(UINT32_MAX);
    std::fill(cnts.begin(), cnts.end(), false);
    static std::vector<uint32_t> cpy;
    cpy.resize(len + 1);
    for (size_t i = 0; i < len; i++)
        cnts[data[i]] = true;
    for (size_t i = 0, j = 0; i < cnts.size(); i++) {
        cpy[j] = i;
        j += cnts[i];
    }
    std::copy(cpy.data(), cpy.data() + len, data);
}
```

Appendix B

FPU-accelerated deunification in spaced out mapping with dynamic padding

Before we realized that the level-order index of the unified addresses into the occurence list with size N was simply contained as a big-endian number in the most significant $\lceil \log_2(N+1) \rceil$ bits of the address, we used a different way of providing $O(1)$ deunification performance in the spaced out mapping with dynamic padding provided in Chapter 4. We include it here, not as a serious alternative to the approach presented in the main thesis text, but because it shows an esoteric use case for the x87 Floating Point Unit.

Instead of keeping the entries in level-order in the occurrence list, they can also be kept in ascending order of their unified values, called the in-order iteration order of the tree, essentially reducing the list to a complete binary tree with the amount of levels $L = \lceil \log_2(n+1) \rceil$ for n the size of the occurrence list. This allows a binary search for the correct logical allocation with a complexity of $O(\log(n))$: from most-significant to least-significant bit, the unified value contains the choice that should be made on every node in the tree to get to the entry that contains its logical allocation. For example, an offset pointer into the purple allocation in Figure 4.8 starts with `00...`, corresponding to the 4th element in the breadth-first traversal of the tree. The first `0` in the unified value rules out indices $> size/2$, as their unified values are guaranteed to be bigger than `0...`. The second `0` means that the index is $< size/4$, as entries with an index higher than that start with either `01...` or `11...`. Together these constrain the index to be 0 for an occurrence list of size 4, which is where the purple allocation resides. To clarify, a practical implementation of the procedure is shown implemented in C in Listing B.1.

```
size_t unified2idx(uintptr_t unified_val, size_t nr_entries) {
    double idx = 0;
    unsigned char levels = ceil(log2(nr_entries + 1));
    for (unsigned char i = 1; i <= levels; i++) {
        bool greater = (((uint64_t)1) << (64 - i)) & unified_val;
        idx += ((double)(greater * nr_entries)) / (i + 1);
    }
    return floor(idx);
}
```

Listing B.1: Calculating the index of a logical allocation in the occurrence list using a binary search based off the unified value in the spaced out mapping with dynamic padding.

The implementation in Listing B.1 essentially considers the unified value as a sequence of booleans that decide whether an exponentially decreasing positive value gets added to the index or not. This is represented visually in Figure B.1[1]. The formula for calculating the index is thus

$$index = \left(b_{63} * \frac{S}{2} + b_{62} * \frac{S}{4} + b_{61} * \frac{S}{8} + ... + b_{64-L} * \frac{S}{2^L} \right) = S * \sum_{i=1}^{L} (b_{64-i} * 2^{-i})$$

with $b_{63...64-L}$ the L most significant bits in the unified value and S the size of the occurrence list. Essentially, the above formula specifies a mathematical relation between the element value and its in-order index in this specific complete binary tree: the *multipliers* are multiplied by their respective bit value and the size of the occurrence list and summed together to produce the index value of the corresponding entry in the occurrence list.

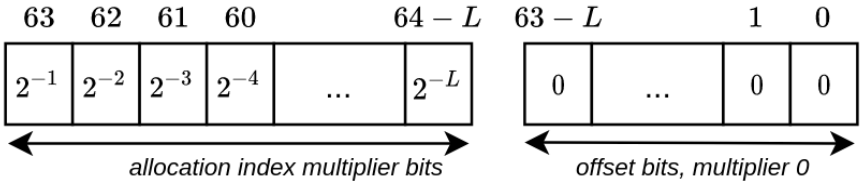


Figure B.1: Multiplier values of every bit in a unified pointer value using the spaced out mapping with dynamic padding.

This operation closely resembles the encoding of floating point values as defined by the IEEE 754 standard [78]. A double-precision floating point value is subdivided into 3 areas as represented by Figure B.2

[1] Note that only the most significant L bits have non-zero multiplier values, as the binary tree does not contain every 64-bit value in the general case.

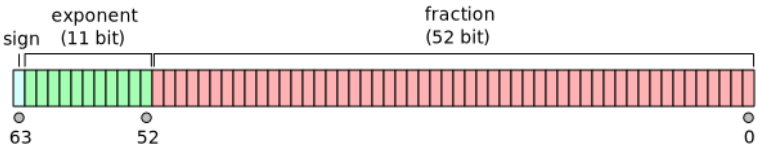


Figure B.2: "The memory format of an IEEE 754 double floating point value.", source: [3].

and the decimal value is computed as follows

$$(-1)^{sign} * \left(1 + \sum_{i=1}^{52} (f_{52-i} * 2^{-i})\right) * 2^{exponent-1023}$$

where $f_{52...1}$ are the fraction bits. The summation term contains exactly the behaviour needed: a series of bits is multiplied with an exponentially decreasing negative power of 2 and the results are summed together.

When filling in the parameters, the sign bit should clearly be 0. The 51st fraction bit is multiplied by 2^{-1}, wich is desired, so the 52 fraction bits can be set to the 52 most significant bits of the unified pointer value[2]. One would think that the exponent can be set to $L + 1023$ such that $2^{exponent-1023} = size$. However, this only works when the size is a power of 2, as L is ceiled. Therefore, the multiplication should happen separately, after converting the **`double`** back to an integer. The exponent is thus set to 1023, which renders the exponent term 1. After converting to an integer, the result will be $1 + sum\_of\_multipliers$, so to get the desired result, it should be multiplied by $size$ after subtracting 1. The resulting implementation is shown in Listing B.2.

The Floating Point Unit (FPU), ubiquitous inside many modern processors, essentially performs the binary search in a single instruction this way, further decreasing deunification time complexity from $O(log(n))$ to $O(1)$.

[2]Note that the use of 64-bit floating point values restricts the binary tree to 52 levels, which is no problem in practice as the x86-64 architecture currently limits the addressable region to 52 bits in theory, and 48 bits in practice.

```
typedef union number {
    double dub;
    struct {
        uint64_t fraction : 52;
        uint16_t exp : 11;
        bool sign : 1;
    } __attribute__((packed)) data;
} number;

size_t unified2idx(uintptr_t unified_val, size_t nr_entries) {
    number idx;
    idx.data.sign = 0;
    idx.data.exp = 1023;
    uint64_t levels = ceil(log2(nr_entries + 1));
    idx.data.fraction = ((unified_val >> 12) >> (52 - levels));
    idx.data.fraction <<= (52 - levels);
    idx.dub = floor((idx.dub - 1)*nr_entries);
    return idx.dub; // conversion to integer
}
```

Listing B.2: Calculating the index of a logical allocation in the occurrence list using a hardware-accelerated binary search based off the unified value in the spaced out mapping with dynamic padding.

Appendix C

Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel

Over het tegengaan van adres-gevoelig gedrag in MVUOs

Adriaan Jacobs
KU Leuven
Gent, Belgium
adriaan.jacobs@student.kuleuven.be

Ruben Mechelinck
imec-DistriNet, KU Leuven
Gent, Belgium
ruben.mechelinck@kuleuven.be

Jonas Vinck
imec-DistriNet, KU Leuven
Gent, Belgium
jonas.vinck@kuleuven.be

Alexios Voulimeneas
imec-DistriNet, KU Leuven
Gent, Belgium
alex.voulimeneas@kuleuven.be

Stijn Volckaert
imec-DistriNet, KU Leuven
Gent, Belgium
stijn.volckaert@kuleuven.be

SAMENVATTING

Multi-variante Uitvoeromgevingen (MVUOs) beloven uitgebreide bescherming tegen aanvallen op basis van geheugenfouten door meerdere gediversifieerde varianten van hetzelfde programma parallel uit te voeren, waarbij de toestand van het programma op bepaalde Rendez-Vous Punten (RVPs) wordt gecontroleerd en vergeleken. Bestaande programma's kunnen zich echter van nature niet-deterministisch gedragen wanneer ze worden onderworpen aan bepaalde vormen van diversificatie die een typische MVUO of besturingssysteem toepast. Een opmerkelijk probleem is Adres-Gevoelig Gedrag (AGG), waarbij specifieke adreswaarden de *control flow* van een programma beïnvloeden of de staat ervan op het moment van de RVP's aanpassen, wat goedaardige divergenties in de MVUOs veroorzaakt. Hoewel dit de toepasbaarheid van MVUOs in de praktijk beperkt, bestaat er geen eerder werk dat de volledige reikwijdte van dit probleem verkent. Dit werk beoogt die leemte op te vullen en onderzoekt de verschillende manieren waarop AGG wordt uitgeoefend door programma's in de praktijk.

We presenteren een overzicht en categorisatie van het probleem om de oorzaken en gevolgen ervan te belichten, alsmede om een theoretisch referentiekader te scheppen. Bovendien onderzoeken we in detail de mogelijkheid om het AGG van een programma automatisch te verwijderen of te neutraliseren met behoud van maximale diversificatie. We stellen verschillende strategieën voor die we evalueren op basis van criteria zoals doeltreffendheid, veiligheidsimpact en prestatie-impact. Sommige van de strategieën in dit werk kunnen alle door ons onderzochte voorbeelden van AGG op transparante wijze onderdrukken.

Trefwoorden: Niet-determinisme, Multi-Variante Uitvoeringsomgevingen, Geheugenlayoutdiversificatie, Adres-gevoelig Gedrag

1 INTRODUCTIE

Multi-variante uitvoeringsomgevingen (MVUOs) zijn een een veelbelovend onderzoeksgebied in de strijd tegen binaire exploitatie-aanvallen, door de uitgebreide variëteit aan aanvallen waartegen ze bescherming bieden en de lage prestatie-impact waarmee ze dat doen. In een MVUO worden N varianten van hetzelfde programma gelijktijdig in *lockstep* uitgevoerd, onder toezicht van een monitor die ze dezelfde invoer geeft en de uitvoering beëindigt als ze van elkaar afwijken. De varianten worden met behulp van softwarediversiteitstechnieken uit hetzelfde programma gegenereerd, in die mate dat het moeilijk is voor een aanvaller om een kwetsbaarheid in het programma te exploiteren op een manier die de varianten niet van elkaar doet afwijken. Typisch onderschept een monitor de varianten voor elke systeem-oproep die ze uitvoeren [1].

Sommige van de softwarediversiteitstechnieken die door MVUOs worden versterkt diversifiëren de geheugenlayout van de varianten. Deze technieken omvatten, onder andere, het gebruik van verschillende geheugenallocatie-algoritmen [2, 3], het omkeren van de groeirichting van de stack [4, 5] en Address Space Layout Randomization (ASLR) [6]. Echter, eerder onderzoek heeft aangetoond dat programma's zich niet deterministisch gedragen bij een veranderende geheugenindeling [1, 7]: ze kunnen verschillende resultaten opleveren, verschillende codepaden volgen of verschillende neveneffecten hebben, afhankelijk van waar het geheugen dat ze gebruiken zich in de adresruimte bevindt. We noemen dit het Adres-Gevoelig Gedrag (AGG) van een programma, en aangezien een MVUO varianten die niet dezelfde systeem-oproepen met equivalente argumenten uitvoeren als kwaadaardig beschouwt, veroorzaakt dit veel goedaardige divergenties bij software in de echte wereld. Dit draagt momenteel bij aan de beperkte toepasbaarheid van MVUOs in de praktijk.

2 OVERZICHT VAN ADRES-GEVOELIG GEDRAG

De meeste prominente manier waarop programma's typisch adresgevoelig zijn is door adreswaarden te printen of als sleutel in hashtabellen te gebruiken. In het eerste geval zal de monitor een verschil bemerken in de argumenten van de varianten voor de `write` systeem-oproep, in het laatste geval zal de interne volgorde waarin de adreswaarden opgeslagen zijn in the hashtabel en hun verdeling over de *buckets* verschillen in de varianten. Wanneer het programma daarna itereert over de hashtabel zal het de adreswaarden in een verschillende volgorde beschouwen in de varianten, waardoor die compleet verschillende acties kunnen ondernemen in een verschillende volgorde.

Een ander voorbeeld is het gebruik van niet-geïnitialiseerd geheugen door het programma. Als variabelen of geheugenregio's niet expliciet geïnitialiseerd worden door de programmeur zal hun inhoud typisch niet dezelfde zijn in de varianten, vooral wanneer die onderworpen zijn aan een gediversifieerde geheugenlayout.

Als het gebruik van een van deze adres-gevoelige constructies leidt to de uitvoer van niet-equivalente systeemoproepen in de

varianten zal de monitor een divergentie observeren die niet te onderscheiden is van een kwaadaardige divergentie als gevolg van een poging tot exploitatie, en de executie van de varianten stopzetten. Merk op dat adres-gevoelige operaties niet per se tot divergenties leiden: in Listing 1 is te zien hoe het startadres van een string wordt afgerond naar het volgende veelvoud van 8 met behulp van een iteratie.

```
size_t strlen(const char * str) {
    const char *char_ptr;
    for (char_ptr = str;
            (char_ptr & (7u)) != 0;
            ++char_ptr)
        if (*char_ptr == '\0')
            return char_ptr - str;
    // ... rest of implementation
}
```

Listing 1: Aligneringsoperatie in de 64-bit `strlen` implementatie in *glibc*.

Hoewel de loop conditie adres-gevoelig is door de waarde van `char_ptr` te inspecteren, worden er geen systeem-oproepen uitgevoerd in dit fragment en treedt er dus geen divergentie op. Dit noemen we *veilig* adres-gevoelig gedrag, in tegenstelling tot *onveilig* AGG dat wel tot divergenties leidt.

3 MITIGATIE VAN ADRES-GEVOELIG GEDRAG

Aangezien vals-positieve divergenties als gevolg van adres-gevoelig gedrag niet te onderscheiden zijn van echte divergenties veroorzaakt door varianten die aangevallen worden, menen we dat de oplossing van het probleem erin bestaat om te voorkomen dat de varianten überhaupt afwijkend gedrag vertonen ten gevolge van AGG. Dit doen we door een nieuwe adresruimte te introduceren in alle varianten genaamd de *eengemaakte adresruimte* (EAR) die elke variant mapt naar zijn gediversifieerde, variant-specifieke adresruimte (VSAR). Deze mapping gebeurt op een allocatie-granulariteit, aangezien operaties die ondersteund zijn op variant-specifieke adressen zoals *pointer arithmetic* ook ondersteund moeten zijn in de EAR. Echter, niet alle operaties kunnen op deze manier ondersteund worden: een operatie die het geheugen op een bepaald adres leest of ernaar schrijft is enkel valide voor variant-specifieke adreswaarden, aangezien de eengemaakte adreswaarden nooit naar equivalente plaatsen in het geheugen van de varianten kunnen refereren door de geheugenlayoutdiversifatie. Voor zulke operaties moeten alle adreswaarden in het programma dus ten minste tijdelijk vertaald worden naar hun variant-specifieke representatie.

In het algemeen moet er voor de neutralisatie van AGG een set van regels opgesteld worden die bepalen wanneer adreswaarden bestaan in hun eengemaakte vorm en wanneer niet. Deze moeten er enerzijds voor zorgen dat, wanneer varianten het adres gebruiken om geheugen te manipuleren, het in zijn variant-specifieke vorm voorkomt, en wanneer het gebruikt wordt in adres-gevoelige operaties, het in zijn eengemaakte vorm voorkomt om een gelijke uitkomst van die operaties in de varianten te garanderen. Deze set van regels vormt in feite een *eenmakingsgrens* in alle varianten, die bepaalt wanneer adreswaarden in welke vorm bestaan. We analyseren verschillende posities waar we deze grens kunnen leggen.

3.1 De ligging van de eenmakingsgrens bepalen

De eenvoudigste positionering van de eenmakingsgrens is om adreswaarden standaard altijd in hun eengemaakte vorm te laten bestaan, en enkel tijdelijk te vertalen naar hun variant-specieke vorm wanneer ze gebruikt worden om geheugeninhoud te manipuleren. Aangezien operaties die geheugen manipuleren nooit onveilig kunnen zijn, i.e. ze behandelen altijd dezelfde geheugeninhoud in de varianten zolang er geen eerdere divergentie heeft opgetreden, zorgt dit er triviaal voor dat alle adres-gevoelige operaties in de varianten op exact dezelfde waarden opereren, en dus hetzelfde effect hebben. Hiernaar refereren we als de *pointer-granulaire* eenmakingsgrens.

Concreet betekent dit dat alle manieren waarop het programma in de eerste plaats aan adreswaarden kan geraken geïnstrumenteerd moeten worden om de eengemaakte adreswaarden te voorzien in de plaats, en dat alle `load`/`store` operaties geïnstrumenteerd moeten worden om het adres waarop ze opereren te vertalen naar de VSAR. We identificeren 3 bronnen van adreswaarden in C/C++ programma's: *address-of* operaties (& operator), calls naar dynamische allocatiefuncties wiens operatie gediversifieerd is en decay-operaties van arrays of functies naar pointer variabelen. Als deze allemaal geïnstrumenteerd worden, kunnen er geen divergenties meer optreden ten gevolge van adres-gevoelig gedrag.

Echter, deze eenmakingsgrens houdt geen rekening met de reden waarom geheugenlayoutdiversificatie wordt toegepast in de eerste plaats: ongeldige geheugentoegangen zoals *out-of-bounds* `load`/`store` operaties [8] worden verondersteld verschillende effecten te hebben in de varianten, zodat een aanval geen gadgets kan lokaliseren in alle varianten tegelijk. Als pointer variabelen standaard eengemaakt zijn zal alle *arithmetic* op deze variabelen ook in de eengemaakte adresruimte gebeuren. Daardoor kan het resultaat van out-of-bounds arithmetic zich in een regio van de EAR bevinden die overeenkomt met een andere allocatie, gelijkaardig aan hoe een *buffer overflow* in de VSAR de geheugeninhoud van een volgende allocatie manipuleert. Elke geheugentoegang op deze out-of-bounds eengemaakte waarde wordt echter *by definition* vertaald naar equivalente geheugenregio's in de varianten en werkt zo de diversificatie van geheugenlayout tegen. Merk op dat dit niet alle geheugenlayoutdiversificatie neutraliseert: enkel `load`/`store` operaties die expliciet in de source code voorkomen moeten geïnstrumenteerd worden, aangezien enkel adreswaarden die in de source code voorkomen eengemaakt zijn. Het return adres in een functie frame dat bepaalt naar waar de controle terug moet springen wanneer de functie returnt wordt bijvoorbeeld niet expliciet geschreven in source code, dus dat adres zal altijd variant-specifiek zijn. De geheugentoegang die gebeurt wanneer de functie returnt is ook niet geschreven in source code en daarom niet geïnstrumenteerd. Deze eenmakingsgrens behoudt daardoor bescherming tegen klassieke *stack smashing attacks* [9]. De geheugentoegang die gebeurt door het oproepen van functies via functie pointers is echter wel geïnstrumenteerd, aangezien de functiepointers eengemaakt zijn. Dit zorgt ervoor dat de aanvaller een invalide geheugentoegang zoals de out-of-bounds check can exploiteren om de waarde van een functiepointer te wijzigen naar een eengemaakte waarde

die vertaalt naar dezelfde functie in al de varianten. Simpelweg een uitzondering maken voor functiepointers en deze wél in variant-specifieke vorm laten bestaan lost dit probleem niet op: enerzijds staat dit de pointer toe om gebruikt te worden in AGG, maar, nog belangrijker, buffer overflows kunnen dan nog steeds gebruikt worden om stringwaarden te overschrijven die later gebruikt worden om dynamisch functies op te zoeken. Een exploit in het `sudo` programma [10], gepubliceerd in januari 2021, is hier een voorbeeld van.

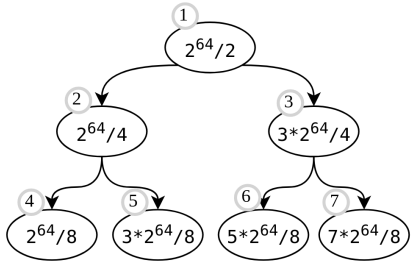
Een alternatieve manier om de eenmakingsgrens te leggen is door alle adreswaarden normaal gezien in hun variant-specifieke vorm te laten bestaan, en pas te vertalen naar eengemaakte vorm zodra ze geconverteerd worden naar een niet-pointer type, zoals een integer type. Dit zorgt ervoor dat pointer arithmetic niet langer gebeurt op eengemaakte adreswaarden. Echter, de iteratie die leidt tot een out-of-bounds waarde kan ook gewoon gebeuren op de integer representatie van adreswaarden na conversie, met dezelfde consequenties als bij de pointer-granulaire eenmakingsgrens tot gevolg. Bovendien omvat deze eenmakingsgrens niet alle AGG: het verschil tussen twee variant-specifieke adreswaarden is ook een variant-specifieke waarde en kan gebruikt worden om divergenties te veroorzaken.

Ons onderzoek suggereert dat zodra een variabele is vertaald naar zijn eengemaakte vorm, er in principe geen operatie op mag gebeuren waarvan het resultaat mogelijks een out-of-bounds eengemaakte adreswaarde is. In het algemeen beperkt dit de set van toegestane operaties op eengemaakte waarden tot diegenen die de waarde transformeren op een irreversibele manier, zodat het programma het resultaat nooit kan gebruiken om een variant-specifiek adres te bekomen. Indien dat inderdaad nooit meer gebeurd, is er geen nood meer aan de optie om eengemaakte adressen terug te vertalen naar variant-specifieke adressen: elke operatie die een eengemaakt adres vereist mag nooit op basis van dat adres een vertaling naar de VSAR ondergaan en gebruikt worden om geheugen te manipuleren aangezien het in eengemaakte vorm out-of-bounds heeft kunnen gaan. Dit maakt het moeilijk om het fragment in Listing 2 te ondersteunen vanuit een AGG perspectief.

```
std::vector<int32_t> rels;
char* b_ptr = malloc(sizeof a_struct);
size_t N = 10;
for (int i = 0; i < N; i++)
  rels.push_back(malloc(sizeof a_struct)
                    - b_ptr);
std::sort(rels.begin(), rels.end());
std::vector<a_struct*> abs_ptrs;
for (auto rptr : rels) {
  a_struct* abs_ptr = b_ptr + rptr + argc;
  *abs_ptr = { /* untrusted data */ };
  printf("%p\n", abs_ptr);
}
```

Listing 2: Een voorbeeld van AGG op basis van relatieve pointers.

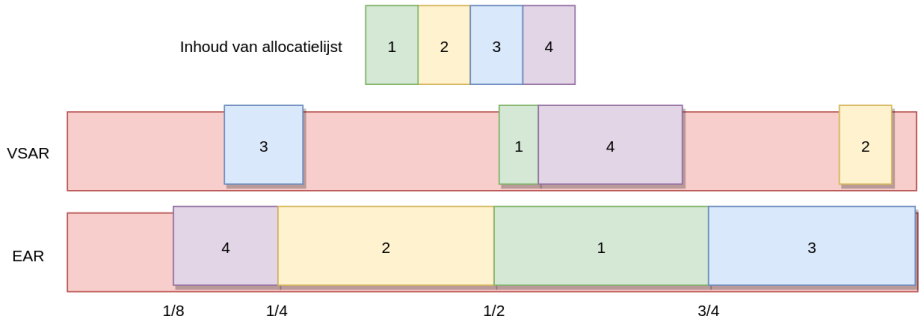
Als de waarden in `rels` eengemaakt zijn, i.e. het verschil tussen twee eengemaakte adreswaarden representeren, moet de reconstructie van `abs_ptr` eerst `b_ptr` vertalen naar een eengemaakte waarde, dan `rptr + argc` eraan toevoegen en dan terug vertalen naar de VSAR. Dit is dus niet veilig indien `argc` een onbetrouwbare input is. Echter, indien de waarden in `rels` niet eengemaakt zijn zal de `std::sort` operatie ervoor zorgen dat de adreswaarden in een verschillende volgorde uitgeprint worden, wat tot een divergentie leidt. Ons onderzoek verschaft geen volledige oplossing voor dit probleem.



Figuur 1: Breedte-eerst iteratie over de eerste 3 niveaus van een gebalanceerde binaire boom van alle 64-bit getallen.

3.2 Het ontwerp van de eengemaakte adresruimte

Om de mapping tussen variant-specifieke en eengemaakte adreswaarden te implementeren baseren we ons op de kennis dat alle allocaties in dezelfde volgorde gebeuren worden in de varianten. We kunnen daarom het beginadres van elke allocatie, zowel statisch als dynamisch, registreren in een lijst en de positie in die lijst gebruiken om op een variant-agnostische manier naar dezelfde allocatie in alle varianten te refereren. Idealiter willen we er ook voor zorgen dat wanneer een eengemaakte adreswaarde in allocatie A X aantal bytes out-of-bounds gaat en dan terug vertaald wordt naar de VSAR, de variant-specifieke adreswaarde die we verkrijgen X aantal bytes out-of-bounds is van de equivalente allocatie A in de VSAR. Dit zorgt ervoor dat een out-of-bounds geheugentoegang nog steeds verschillende effecten heeft in de varianten. Hoewel we dit niet kunnen ondersteunen voor eengemaakte adreswaarden die willekeurig ver out-of-bounds zijn, kunnen we wel de eengemaakte beginadressen van de allocaties zo veel mogelijk spreiden over de EAR en er dan vanuit gaan bij een vertaling van een eengemaakte waarde P naar de VSAR dat het dichtstbijzijnde eengemaakte beginadres B de allocatie voorstelt waarin P out-of-bounds is. Vanuit de assumptie dat de meeste out-of-bounds waarden voor een gegeven allocatie eerder dicht bij de grenzen van die allocatie liggen kunnen we dan de bulk van de out-of-bounds waarden op een veilige manier terug vertalen naar de VSAR. Om deze spreiding te bekomen willen we idealiter over alle 64-bit waarden itereren op een manier die de waarden in de meest uiteengespreide volgorde bezoekt, zonder dezelfde waarde tweemaal te bezoeken: eerst het middelste adres in de hele 64-bit ruimte, dan een adres een kwart erin, dan driekwart erin, dan een achtste, etc. Daartoe beschouwen we een hypothetische gebalanceerde binaire boom van alle 64-bit waarden,



Figuur 2: Conceptuele layout van de mapping tussen de VSAR en de EAR.

waarvan de bovenste 3 niveaus gerepresenteerd zijn in Figuur 1. Een breedte-eerst iteratie over deze boom passeert alle 64-bit waarden in de gewenste volgorde. Indien we elke allocatie, onafhankelijk van zijn grootte, nu toewijzen aan de volgende waarde in de iteratie van de boom, bereiken we een EAR waarin de de allocaties steeds maximaal verspreid zijn over de volledige adresruimte. De mapping van VSAR naar EAR is conceptueel voorgesteld in Figuur 2. Om te vertalen van een eengemaakte adreswaarde naar een variant-specifieke waarde in een mapping met N geregistreerde allocaties kunnen we kijken naar de $\lceil \log_2(N+1) \rceil$ meest significante bits. Deze zullen de breedte-eerst-index van de allocatie waarnaar deze eengemaakte waarde verwijst bevatten in *big-endian* bit volgorde. Door de *endianness* van deze bits om te draaien en dan uit te lezen[1] verkrijgen we dus de gewenste index in $O(1)$.

Als we nu terug het specifieke voorbeeld van Listing 2 beschouwen waarin `argc`, een 32-bit integer, werd opgeteld bij een eengemaakte adreswaarde, kan de resulterende out-of-bounds waarde nooit verder dan 4GB ($= 2^{32}$) buiten zijn allocatie liggen. Voordat deze mapping die out-of-bounds pointer zou vertalen naar een andere allocatie dan degene tot welke hij behoort, en dus een veiligheidsrisico zou vormen, moet de afstand tussen consecutieve allocaties in de EAR 4GB of minder zijn. Hiervoor moet de boom 32 levels diep zijn, wat betekent dat meer dan $\sum_{n=1}^{31}(2^{n-1}) = 2^{31} - 1 =$ 2 miljard allocaties moeten geregistreerd zijn. Zolang dat niet het geval is, is dit snippet veilig. Deze mapping voorziet daarom ten minste een stochastische bescherming tegen de veiligheidsproblemen van de EAR.

4 CONCLUSIE

In dit artikel beschouwden we het probleem van adres-gevoelig gedrag (AGG), de invloed die het op dit moment heeft op MVUOs, de vormen waarin het in programma's voorkomt en de mogelijkheden om het op een automatische en transparante manier te neutraliseren. Hiertoe introduceerden we een nieuwe adresruimte waar elke variant een mapping naar heeft vanuit zijn eigen specifieke adresruimte, zodat we variant-agnostische adreswaarden kunnen bezorgen aan adres-gevoelige operaties. We analyseerden de impact van deze mapping op de veiligheidsgaranties die een MVUO beoogt en concludeerden dat deze in sommige gevallen divergenties verbergt van de MVUO. Echter, we toonden ook hoe de kans dat dit gebeurt geminimaliseerd kan worden door een specifieke layout te gebruiken in the variant-agnostische adresruimte.

[1]Op een systeem met *little endian* architectuur, zoals we beschouwen in dit werk.

REFERENTIES

[1] S. Volckaert, "Advanced techniques for multi-variant execution," Ph.D. dissertation, Ghent University, 2015.

[2] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "Kmvx: Detecting kernel information leaks with multi-variant execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 559–572. [Online]. Available: https://doi.org/10.1145/3297858.3304054

[3] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 431–442.

[4] B. Salamat, A. Gal, and M. Franz, "Reverse stack execution in a multi-variant execution environment," in *In Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[5] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: A user space multi-variant execution environment," 2008.

[6] T. Pax, "Pax address space layout randomization (aslr)," *http://pax.grsecurity.net/docs/aslr.txt*, 2003.

[7] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "Ghumvee: Efficient, effective, and flexible replication," in *Foundations and Practice of Security*, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, A. Miri, and N. Tawbi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 261–277.

[8] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106.

[9] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[10] "CVE-2021-3156." Available from MITRE, CVE-ID CVE-2021-3156., January 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156

Appendix D

Poster

Adriaan Jacobs

KU LEUVEN

Faculty of Engineering Technology - Electronics-ICT

Promotor: prof. S. Volckaert

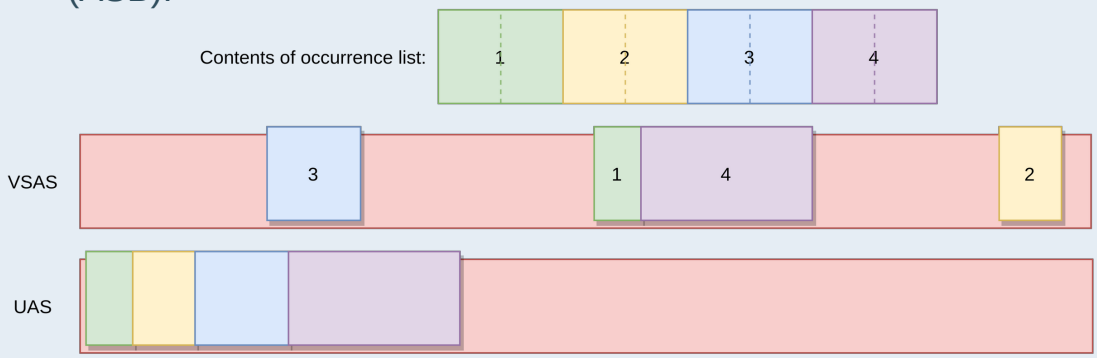
Year: 2020-2021

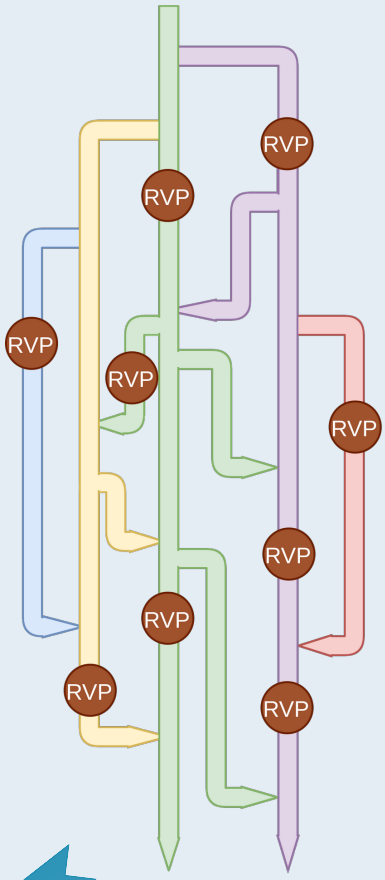
Co-promotors: Jonas Vinck, Ruben Mechelinck, Alexios Voulimeneas

Combating address-sensitive behaviour in MVEEs

PROBLEM

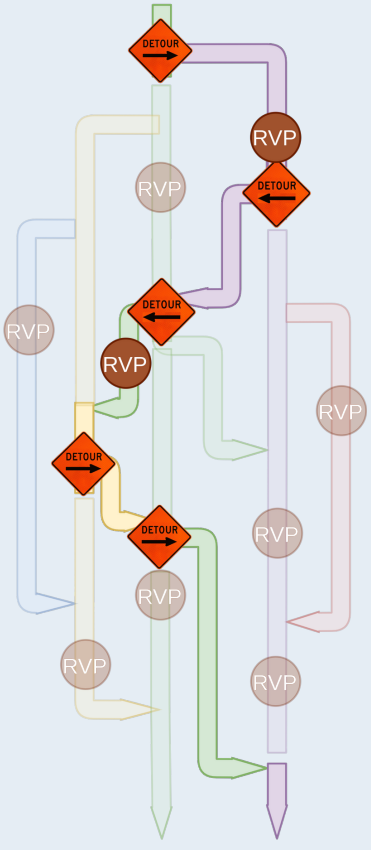
Multi-Variant Execution Environments (MVEEs) execute diversified variants of the same program in parallel and compare their behaviour at certain Rendez-Vous Points (RVPs). Hence, if the behaviour of the program depends on specific address values, the variants might naturally diverge and cause the MVEE to falsely terminate execution. This is called address-sensitive behaviour (ASB).





METHOD

An overlaid, unified address space (UAS) is introduced and mapped to each variant-specific address space (VSAS) with byte-granularity.
Combined with a fine-grained detection approach that highlights potential address-sensitive operations, all ASB can effectively be neutralized across variants by occasionally and temporarily replacing pointer values with their variant-agnostic handle at run-time.



RESULTS

The first set of complete categorizations of ASB is presented, exemplified by a large sample of real-world code.
Additionally, multiple automatic mitigation strategies are explored and analyzed.
Finally, an LLVM-based solution is implemented and evaluated on the GNU core utilities.

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
TECHNOLOGIECAMPUS GENT
Gebroeders De Smetstraat 1
8200 GENT, België
tel. + 32 50 66 48 00
iiw.gent@kuleuven.be
www.iiw.kuleuven.be