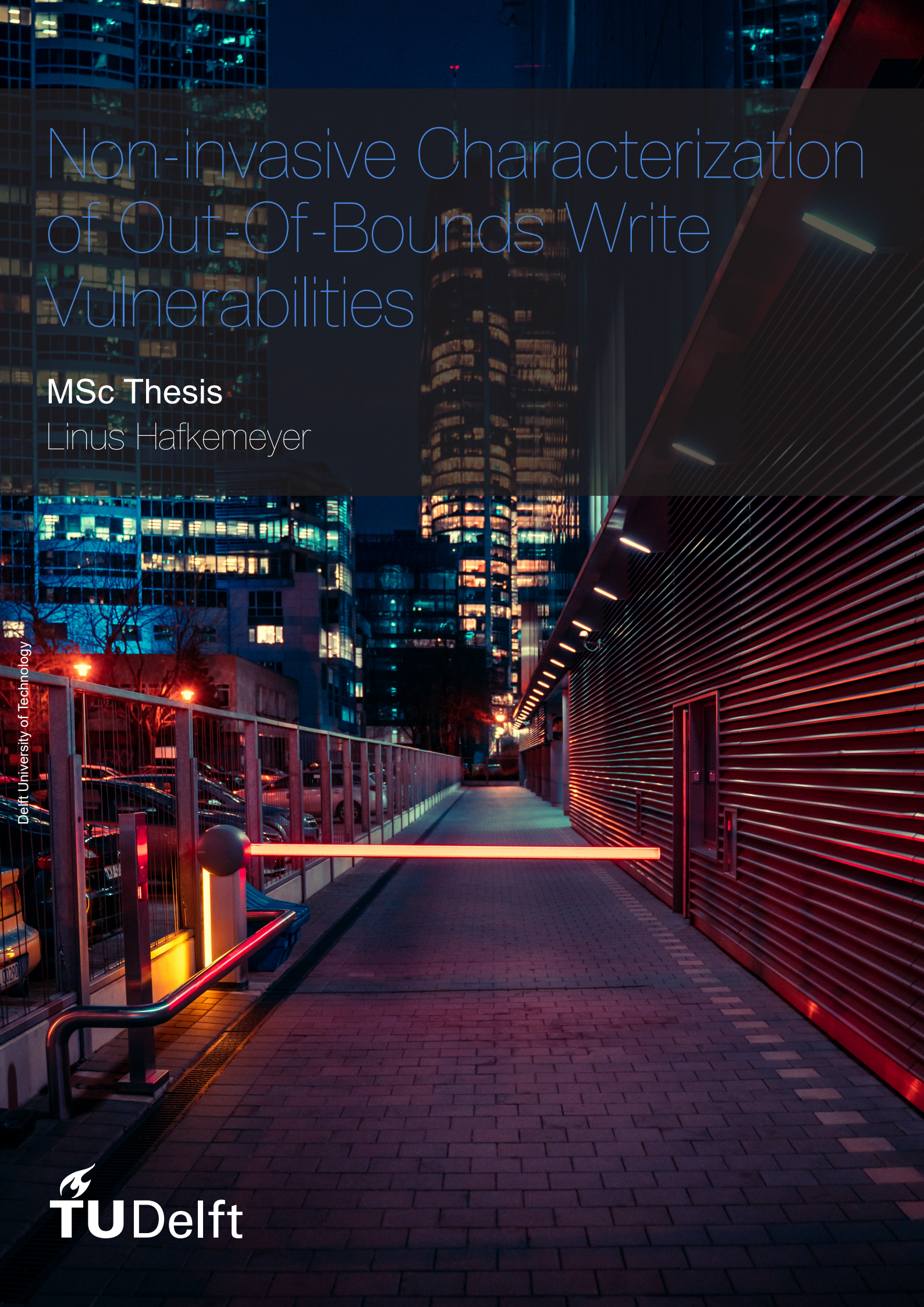
Non-invasive Characterization of Out-Of-Bounds Write Vulnerabilities

MSc Thesis

Linus Hafkemeyer



Delft University of Technology

TU Delft

Non-invasive Characterization of Out-Of-Bounds Write Vulnerabilities

by

Linus Hafkemeyer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday June 30, 2022 at 10:00.

Student Number:	5365902	
Project Duration:	November 15, 2021 - June 30, 2022	
Thesis Committee:	Dr. Andrea Continella,	University of Twente, supervisor
	Dr. Sicco Verwer,	Delft University of Technology
	Dr. Soham Chakraborty,	Delft University of Technology

Cover image by Adam Borkowski

An electronic version of this thesis is available at `https://repository.tudelft.nl`.

TU Delft

Abstract

As more and more aspects of our society and economy rely on software, security vulnerabilities in programs have become an increasingly significant threat. One such class of vulnerabilities are out-of-bounds writes, which are still one of the most widespread and dangerous types of security bugs in memory-unsafe programming languages five decades after their initial conceptualization.

Their discovery through automated methods like fuzzing, however, has gained substantial traction in the past years and facilitates their discovery in large numbers with minimal human intervention. On the other hand, the analysis following their initial discovery - triaging, root cause analysis, and patching - mostly requires human expertise and intuition. As this need for manual effort affects the time to assess a vulnerability's security implications and prioritize patching accordingly, partial automation of the triaging process is essential. A promising approach is the automatic extraction of vulnerability characteristics that provide vital clues for a bug's exploitability. Such characteristics, which typically require substantial effort to collect manually, can aid a human analyst in triaging and thereby speed up the process while, at the same time, making it more accessible.

In this work, we investigate how the set of affected source code-level objects, which is a decisive indicator of an out-of-bounds write vulnerability's exploitability, can be automatically distilled from a program and an input suspected of causing an out-of-bounds write. As this poses unique challenges with regard to invasiveness, we propose a novel approach for out-of-bounds write detection that facilitates monitoring a compiled program for spatial memory safety violations without the need for instrumentation. We implement a prototype of our design and evaluate it on benchmarks and real-world vulnerabilities, showing that its detection performance is largely on par with state-of-the-art instrumentation-based detection approaches and even outperforms them in several scenarios at the cost of increased performance overhead and a higher rate of false positives.

Preface

Before you lies the Master's thesis that, after seven months of hard work, marks the end of my two years at the TU Delft, as well as the end of my 17 years as a student. Looking back, I am amazed by the twists and turns this project took from the initial idea around a year ago to the finished thesis. Although it turned out very different than initially expected, I am happy and proud to present to you the final results of my thesis on *Non-invasive Characterization of Out-Of-Bounds Write Vulnerabilities*.
I would like to thank a number of people who - directly or indirectly - had a determining influence on this thesis.

First of all, I would like to thank my supervisor Andrea Continella for all his guidance, encouragement, and motivating words throughout the entire journey over the past year. Without all his support and his willingness to supervise me despite studying at a different university, I most certainly could not have written my thesis about such a topic. Furthermore, I would like to thank my official TU Delft supervisor Sicco Verwer for making my supervision by Andrea possible and being part of my thesis committee, as well as Soham Chakraborty for his willingness to be part of my committee.

Next, I would like to thank Armin, Aslı, Charlie, Egon, Luke, Marco, Massi, Roland, and Sven for all the nice coffee and lunch breaks and for accompanying me throughout the past years of my studies. Thanks for all the nice talks, the mutual support and discussion, and for all the great group projects without which the past two years certainly would not have been as pleasant.

Finally, I would like to thank my friends and family for all their support during my studies, the good times, as well as for bearing with me during the more difficult times of the past months. I especially want to thank Luna for her endless support and patience during the most challenging parts of my thesis.

Linus Hafkemeyer
Delft, June 2022

Contents

1 Introduction

Software is everywhere. Billions of interconnected devices, all of which run some more or less complex software, affect nearly every aspect of our daily lives – directly or indirectly. Thus, it is no understatement to say that our society is heavily dependent on software. However, software is in most cases also created by humans, making it susceptible to human error, which in some cases manifests itself in security vulnerabilities. The issue is amplified by the fact that even nowadays, two of the most popular programming languages – C and C++ – are not memory safe [1]. This opens the door to a hazardous category of software flaws – memory bugs. One of the most notorious vulnerabilities falling into this category are out-of-bounds (OOB) writes. Almost 50 years after the first recorded mention of a buffer overflow exploit [2], and 34 years after the first documented malicious exploitation of a buffer overflow by the Morris worm, OOB writes are still regarded as one of the most dangerous types of software vulnerabilities [3]. This is not only due to their unabated abundance in modern-day software but also due to the comparatively high reliability of corresponding exploits.

Over the years, a vast number of defenses against OOB writes and other memory corruption bugs have been proposed. Preventive approaches like the removal of notoriously unsafe functions from standard libraries [4], and especially memory safe, yet fast, programming languages [5], [6] will likely continue to improve the situation but are incapable of solving the problem without full adaption, including in legacy software.

Mitigation approaches aimed at complicating or preventing exploitation of memory corruption bugs [7], [8] have been proposed and introduced into operating systems and compilers in large numbers. Although they are often successful in making exploitation more tedious and difficult, attackers continue to find ways and develop new techniques for evading them.

Detection approaches such as static program analysis and particularly dynamic program analysis using fuzzing or symbolic execution in combination with sanitizers have been hugely successful in discovering a wide range of vulnerabilities [9]. However, as an exhaustive exploration of all possible program paths is infeasible for all but the smallest programs, some bugs will always remain undetected.

Despite the increasing popularity of detection approaches, typically, only a relatively small share of discovered bugs has relevant security implications. Thus, further investigation into the severity of discovered bugs and consequent prioritization for patching is essential. As triaging, root cause analysis, and patching of discovered bugs are usually conducted manually by humans, this is often an expensive and time-consuming process, causing potentially severe vulnerabilities to remain unpatched for a longer time. Therefore, an increase in automation in the process that follows the discovery of a bug can substantially decrease the time that elapses between the initial discovery and the development of a patch, thereby minimizing the period in which systems are vulnerable. However, bug triaging, and especially the assessment of security implications, is often an intricate task that largely relies on human expertise and intuition, making its full automation difficult.

A possible approach to the automation of the bug triaging process is automatic exploit generation (AEG), which has attracted great scientific interest in the last years [10]–[14] and typically aims at analyzing a given vulnerability for its exploitability and attempting to synthesize a corresponding exploit. As a working exploit is an ultimate proof of exploitability, the priority for patching can be set accordingly. Furthermore, an existing exploit is a valuable asset in many scenarios when designing a patch.

However, AEG approaches are still in their infancy. They typically only work for a small set of vulnerability types and have difficulties evading deployed defenses. Thus, AEG solutions cannot keep up with experienced human analysts, and the failure to synthesize an exploit for a vulnerability does not imply unexploitability. Therefore, the expertise of a human is often required nevertheless.

Partial automation of the process, however, is a promising alternative to automation with AEG. By automatically distilling characteristics of a vulnerability that are decisive for its exploitability, a human expert can base their assessment of the bug's security implications on a concise high-level summary of the relevant aspects. This reduces the time-intensiveness of the process and makes the task of determining a bug's security implications more accessible, enabling less experienced analysts to perform it.

Furthermore, such an approach can be used to combat a typical drawback of AEG approaches - their full focus on exploit synthesis, which often causes intermediate analysis results to be of minimal use for a human analyst that investigates the bug either for exploit synthesis after the AEG pipeline failed or for other use cases such as patching. By performing AEG based on automatically extracted human-interpretable vulnerability characteristics, intermediate results can be substantially more helpful for manual analysis.

1.1. Problem Statement

Given their fundamental behavior, a decisive characteristic for the exploitability evaluation of any OOB write vulnerability is the set of memory regions that it affects and their purpose during program execution. For example, writing one byte past the bounds of an array may be entirely harmless if the affected byte serves as padding but could lead to arbitrary code execution in other cases. An example is CVE-2018-6789, in which a one-byte OOB write in the Exim mail transfer agent could lead to remote code execution, bypassing common defenses [15]. An aggregated overview of memory areas affected by an OOB write vulnerability, together with source code-level entities located in these areas, can be a valuable aid for analysts tasked with evaluating the exploitability of a bug, developers tasked with patching it, exploit developers, as well as AEG solutions.

In this work, we investigate how such characteristics of an out-of-bounds write vulnerability can be automatically distilled from a program. In contrast to similar state-of-the-art research in the field of memory sanitization, we aim to extract capabilities that are truthful with regard to the form in which the program under test is deployed in practice, thereby providing results on vulnerability characteristics that are relevant not only in laboratory settings but for the real-world usage of the program. Furthermore, we aim to keep the resulting information on vulnerability characteristics universal and exploitation-agnostic to facilitate the integration of our approach into other pipelines, e.g., for automatic exploit generation or more sophisticated vulnerability analysis.

We envision to realize this by developing a system that takes a program under test together with an input to it that is suspected of causing an OOB write and performs fine-grained monitoring for out-of-bounds write vulnerabilities while executing the program, mapping affected memory regions to the corresponding source code-level entities and reporting the results in a concise and easily interpretable form. This poses several challenges:

Invasiveness: Existing tooling for detecting OOB writes usually performs heavyweight instrumentation that modifies the memory layout and runtime behavior of the program under test and thus potentially prevents insights gathered about the modified program from applying to the original program.

Loss of source code semantics: The compilation to machine code causes large parts of information on source code semantics to be lost. Such information includes the structure of source code-level entities like variables and data types, as well as information on which entity a specific write to memory is intended to modify according to the program semantics. However, this type of information is vital for facilitating the detection of OOB writes. Furthermore, it is essential for achieving easy interpretability of the results by mapping the memory regions affected by an OOB write vulnerability to their corresponding source code entities.

Optimized code: Modern compilers can perform sophisticated optimizations during compilation to decrease the program's execution time, storage size, or memory footprint. Such optimizations commonly cause substantial modifications to the program's machine code-level structure and memory layout, requiring low-level dynamic program analysis approaches to handle a large variety of different machine code patterns and structures. As we aim to analyze out-of-bounds write vulnerabilities as they exist in production builds of programs, handling optimized code is essential.

To keep this project achievable, we limit the scope to the characterization of OOB write vulnerabilities in programs written in the C programming language, compiled with the Clang compiler for Linux on AMD64 platforms. Furthermore, we only aim to characterize OOB write vulnerabilities occurring on the stack and in the global memory sections. An important reason for choosing the stack over the heap is generalizability. The stack is used and maintained in almost the exact same fashion across different architectures and operating systems. At the same time, heap management differs significantly not only between architectures and operating systems, but also across different implementations of the C standard library. Furthermore, as the stack generally stores data that directly affects the program's control flow, its corruption is often likely to have security implications.

1.2. Research Questions

The objective of this research project is to propose an approach for performing non-invasive detection of out-of-bounds write vulnerabilities that facilitates the identification of affected source code-level entities in memory. As such, we formulate the following research question:

How can the effects of out-of-bounds write vulnerabilities on source code-level objects in uninstrumented compiled programs be determined through automatic analysis?

The research question can be decomposed into the following sub-questions:

SQ1: How can out-of-bounds writes within different program sections be detected without invasive modification of the program?

SQ2: How can the affected regions in memory be mapped to primitive types and composite structures of the source code?

SQ3: How does a non-invasive out-of-bounds write detection approach perform at different levels of optimization applied by the compiler?

1.3. Contributions

In this work, we propose a new approach for the dynamic detection of OOB write vulnerabilities in C programs. Contrary to existing works, our approach does not modify the compiled program by means of compile time instrumentation and, as such, is entirely non-invasive. This non-invasiveness poses several unique challenges, some of which we solve by providing a conceptual framework for making source code-level semantic information available to our low-level OOB write checking approach.

An implementation of our approach achieves a detecting rate of around 95% in the RIPE benchmark [16], compared to the 70% and 34% achieved by the instrumentation-based current state of the art approaches AddressSanitizer and SoftBound. The cost of this is a substantially increased running time overhead, along with an increased chance of false positives.

Our main contributions can be summarized as follows:

- A methodology for overcoming the lack of source code-level semantic information necessary for performing low-level bounds checking by leveraging the intermediate program representation during compilation.
- The first design of a non-invasive OOB write detection approach capable of identifying spatial memory safety violations on the stack and in the global sections.
- A publicly available prototype implementation of our approach design.
- An evaluation of our prototype's performance under laboratory conditions and when exposed to real world vulnerabilities, including a comparison to instrumentation-based state-of-the-art approaches.

1.4. Outline

The rest of this thesis is structured as follows: In Chapter 2 we provide background information on fundamental concepts our work is based on, as well as an overview of the threat landscape our work is concerned with. In Chapter 3 we give an overview of existing work on OOB write detection and summarize fundamental approaches as well as typical strengths and weaknesses. Based on this, we describe the motivation for our design of a non-invasive approach in Chapter 4. Next, we provide a detailed description of our approach's design in Chapter 5, followed by an outline of our prototype implementation in Chapter 6. In Chapter 7 we evaluate our approach's performance from a number of perspectives, including on multiple real-world vulnerabilities, and compare it to existing state-of-the-art approaches. Afterwards, we critically reflect on our work in Chapter 8 and finally provide a conclusion in Chapter 9.

2

Background

This chapter introduces the reader to several fundamental concepts that play a vital role in our work. First, we provide a brief overview of the instruction set architecture on which our approach operates and outline the memory layout of processes on Linux systems.
Next, we provide an overview of the threat landscape our work is concerned with. We describe different types of attacks that we aim to detect with our approach and outline a number of existing countermeasures against such attacks, for which we highlight the shortcomings that lead to memory corruption bugs still being prevalent.
Finally, we provide a brief introduction to the frameworks and platforms our design and implementation are based on.

2.1. AMD64 Instruction Set Architecture

The x86-64 instruction set architecture (ISA), frequently referred to as AMD64, Intel 64, or simply x64, is a 64-bit version of the x86 ISA and has become the most widely used ISA for CPUs in personal computers and servers since its specification in 2000. Although the x86-64 specifications by AMD and Intel are not entirely equivalent, their differences for application-level programming are so minor that they can safely be disregarded. We will hereafter refer to the x86-64 ISA as AMD64, thereby adhering to its original name.

With AMD64 being backwards-compatible to the x86 ISA, it specifies five different operating modes, four of which serve to ensure backward compatibility to 16-bit and 32-bit operating systems and programs. These operating modes differ not only in their address size but also in numerous other aspects such as the general-purpose register width, available register extensions, memory management, and addressing. However, as the 64-bit mode can be considered the dominant operating mode for modern operating systems and programs, we hereafter assume that the CPU always operates in 64-bit mode.

The AMD64 ISA specifies 16 general purpose registers: `rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsi`, `rdi`, `rsp`, and `r8` through `r15`. A number of additional registers are available for specialized tasks, among them floating point registers, the flags register, segment registers, as well as the instruction pointer register `rip` [17].

Being a complex instruction set computer (CISC) ISA, AMD64 contains a large number of instructions, the exact number of which depends on which instructions are considered equal - those with the same mnemonic, the same opcode, or based on another criterion. For the sake of simplicity, we will hereafter use the term *instruction* synonymously with *mnemonic*. Several extensions to the basic AMD64 architecture were specified and implemented in AMD64 CPUs since the original specification of the ISA and include extensions aimed at facilitating parallel data processing (MMX, SSE, AVX-512), introducing additional security features (MPX, SGX), as well as various other purposes. Such extensions commonly cause the addition of new registers, instructions, and concepts but can largely be disregarded for our purposes.

In 64-bit mode, the AMD64 ISA employs a flat virtual address space that does not rely on segmentation. Despite the 64-bit virtual address format, current implementations of the AMD64 ISA restrict the size of the virtual address space by allowing only the lower 48 bits to be used for addressing, which still allows for up to 256 TiB of virtual memory [18].

2.1.1. Memory Addressing

A further consequence of AMD64 being a CISC ISA is that most instructions can directly access memory. Virtual memory locations to be accessed are encoded via the instruction's ModR/M, SIB, and displacement bytes, which facilitate several addressing modes. However, due to the remarkable complexity of these addressing modes, we abstract from them by assuming the existence of the nine addressing modes presented in Table 2.1. This corresponds to an abstraction also used in the official specification [17].

Addressing mode	Example
Displacement	`mov [0x401a20], rcx`
Base	`mov [rax], rcx`
Base + Index	`mov [rbx + r13], rcx`
Base + Displacement	`mov [rax - 0x80], rcx`
Base + Index + Displacement	`mov [rax + rcx + 0x38], rcx`
Base + (Index * Scale)	`mov [rcx + r14 * 8], rcx`
(Index * Scale) + Displacement	`mov [rax * 4 + 0x47f308], rcx`
Base + (Index * Scale) + Displacement	`mov [rcx + r9 * 2 - 0x52], rcx`
`RIP` + Displacement	`mov [rip + 0x7320], rcx`

Table 2.1: Simplified AMD64 addressing modes with examples

Here, both *Base* and *Index* are general purpose registers. *Displacement* is a fixed immediate value up to 8 bytes in size, while *Scale* is a fixed immediate value from the set $\{2, 4, 8\}$.

In practice, there exist complex restrictions on which instructions support which addressing modes and which components can co-exist in an instruction. However, for our purpose, we can safely disregard such restrictions.

2.2. Memory Layout of a Process

The memory layout of a process on an AMD64 Linux system is dictated by the System V ABI [19]. Upon executing an executable file, e.g., by invoking the `execve` system call, the virtual memory is set up for the process by the operating system's loader. This involves creating a number of memory sections in the virtual address space and mapping corresponding sections of the executable file into some of them, most notably the `.text`, `.bss` and `.data` sections. The stack is allocated at the top of the address space but remains largely empty, only populated with arguments and environment variables.

The following important memory sections are commonly found in a process created by executing an ELF (Executable and Linkable Format) file.

text The `.text` section of the process contains the program's executable instructions. It is usually a write-protected copy of the `.text` section in the ELF file and is the only section in this overview that is executable.

data The `.data` section contains the program's initialized static variables, i.e., global variables and static local variables. It is generally readable and writable, as contained data may be modified at run time.

bss The `.bss` section contains the program's uninitialized static variables. It is therefore usually filled with zeroes during program initialization, causing it to use up no space in the ELF file [20]. Like `.data`, it is generally readable and writable.

heap The `heap` section contains memory dynamically allocated at runtime, e.g., via the standard library functions `malloc` and `calloc`. Dynamic memory management is an intricate topic, and many implementations for allocating, freeing, and resizing heap chunks exist. However, although the heap is virtually never one continuous block of memory, it can be said that, on AMD64 systems, the heap grows towards larger memory addresses, as depicted in Figure 2.2.

stack The `stack` is used to store data associated with active subroutines during the program's execution. It is partitioned into non-overlapping continuous chunks of memory called stack frames, each containing data associated with one active subroutine. This data consists of the function's local data, as well as control flow information. Upon calling a function, a new stack frame is created for it below the stack frame of the calling function, causing the stack to grow towards lower memory addresses, as visualized in Figure 2.2.

Figure 2.3 shows the topmost stack layout caused by executing the code in Figure 2.1 after compiling it without optimizations using Clang 13.0.0. Each stack frame starts with the saved instruction pointer (`rip`) for returning to the calling function (blue), as well as the saved base pointer (`rbp`) of the calling function (green). The current function's base pointer usually points to the saved base pointer of the calling function. It is used for accessing automatic variables and function arguments at a relative offset to it. The stack pointer (`rsp`) generally points to the end of the latest allocated stack frame. In the example, the lack of compiler optimizations causes each function to

copy all arguments it receives to the beginning of its stack frame (yellow), followed by space used for automatic variables (purple). The 8-byte alignment of double floating-point values, as well as the required 16-byte alignment of stack pointers dictated by the System V ABI [19] causes padding to be added at multiple locations in this example.

```c
double getEqlTriangleArea(double side_len) {
  double x = sqrt(3) / 4;
  double result = x * side_len * side_len;
  return result;
}

double getShapeArea(double side_len, char type) {
  double area = -1;
  if (type == 'R') {
    area = getSquareArea(side_len);
  } else if (type == 'T') {
    area = getEqlTriangleArea(side_len);
  }
  return area;
}
```

Figure 2.1: C Function Examples

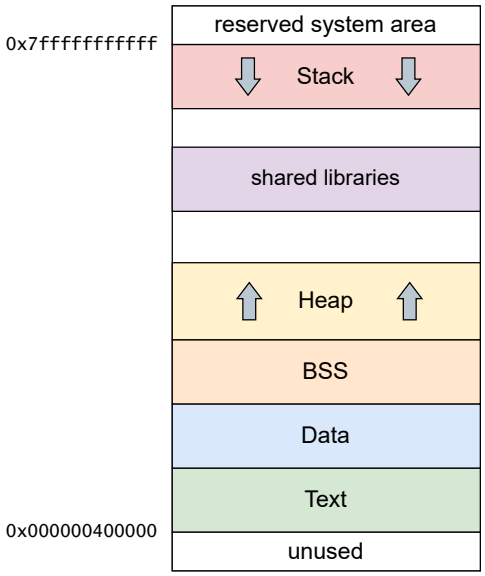


Figure 2.2: Typical memory layout of a process on Linux

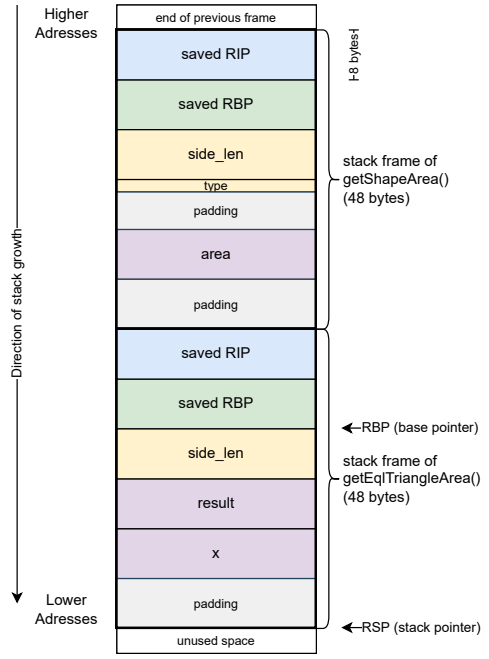


Figure 2.3: Stack layout resulting from function in Figure 2.1

2.2.1. Function Calls

The exact interface used for calling functions on Linux AMD64 systems is dictated by the System V ABI calling convention [19], which includes the procedures for preparation and restoration of the stack, passing arguments and return values, as well as for control flow transfer.

To prepare for a call to another function, the caller must ensure that all arguments to the function are at the locations where the callee expects them. In the System V ABI calling convention, arguments are primarily passed via registers, although the stack is used to pass arguments for which no register is available. The first six arguments of non-floating point type with a size of at most 8 bytes (integers, booleans, pointers, etc.) are passed in the general purpose registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` and the first eight floating-point arguments are passed in the SSE registers `xmm0-xmm7`.

The `call` instruction executed by the caller causes the current instruction pointer to be pushed onto the stack. Upon being called, the first action performed by the callee is to push the current base pointer to the stack, followed by setting the new base pointer accordingly and incrementing the stack pointer to make space for the function's local

data, displayed in Figure 2.4. The callee then executes its actual code and places the return value in `rax` or `xmm0`, depending on the return value type. Finally, the stack frame is torn down again, and execution returns to the caller.

```
push rbp        ; push previous frame pointer onto stack
mov  rbp, rsp   ; set new frame pointer
sub  rsp, 32    ; increment stack pointer
...             ; function logic
add  rsp, 32    ; decrement stack pointer
pop rbp         ; restore previous frame pointer
ret             ; restore previous instruction pointer
```

Figure 2.4: Assembly instructions for setting up and tearing down a stack frame

2.3. Memory Corruption Bugs

High-level programming languages such as Java, C#, and Python are commonly said to be *memory safe* as they employ a number of memory management and safety measures intended to prevent memory corruption bugs. Examples include shielding the raw memory from the programmer by providing them with an abstraction of it, executing error detection checks at runtime, e.g., upon dereferencing pointers, and performing automatic garbage collection. However, programming languages such as C/C++ expose the programmer to the raw memory and leave the task of memory management to them, thereby trading memory safety for efficiency and flexibility. Although developments such as the introduction of smart pointers to the C++ standard library, detection capabilities for certain types of memory corruption bugs in mainstream compilers such as GCC [21], and readily available memory analysis tools such as Valgrind's Memcheck [22] and AddressSanitizer [23] make it easier for developers to find and avoid memory corruption bugs, their prevalence in software is still unabated [3]. In the Chromium project, for example, almost 70% of high-severity security bugs are due to memory safety problems [24].

Memory corruption bugs can generally be divided into two categories - temporal bugs and spatial bugs.

Temporal Bugs Temporal memory corruption occurs when an operation performed on a piece of data stored in memory is invalid at that time, despite being valid at some other point in time. In practice, this usually corresponds to operating on a piece of data outside of its lifetime. Some of the most well-known temporal memory corruption bugs are Use-after-free (UAF) bugs, in which a pointer to a memory chunk on the heap is kept and dereferenced after the chunk has already been freed ("dangling pointer") and potentially been reused for another allocation. In the Chromium project, for example, UAF bugs are responsible for 36% of all high-severity security bugs and more than half of all high-severity bugs caused by memory safety issues [24]. Further examples of temporal memory corruption bugs include Double-free bugs and Invalid-free bugs, which occur when a chunk of allocated memory is freed more than once and when a chunk of memory not previously allocated by a valid memory allocator is freed, respectively.

Although temporal memory corruption bugs tend to be complex to exploit and resulting exploits tend to be less reliable than exploits for spatial memory corruption bugs, they can nevertheless, in many cases, be engineered into full-fledged exploits capable of achieving control flow diversion and even arbitrary code execution [25].

Spatial Bugs Spatial memory corruption occurs when an object in memory is accessed outside of its bounds, either by reading (OOB read) or writing (OOB write)[1].

A typical example of spatial memory corruption in the form of an OOB write is a buffer overflow. Here, a buffer, which can be allocated in the static program sections, on the heap, or the stack, is written to without properly validating that every modified byte will still be within the allocated bounds of the buffer, for example, due to a faulty loop guard. Consequentially, the spatial boundary of the buffer is violated, potentially causing memory allocated for storing other entities to be modified and thereby inducing unexpected program behavior with possible security implications.

While the potential for severe security implications following OOB writes is obvious, OOB reads might seem comparatively innocuous, as they cannot modify the program state. In many cases, however, OOB reads are an essential prerequisite for exploiting other vulnerabilities, as they facilitate information leaks that can be used to circumvent exploitation countermeasures aimed at obfuscating the program's internal representation.

[1] Strictly speaking, OOB reads do not corrupt memory. However, as they are closely related to OOB writes and have similar root causes, we still mention them.

2.3.1. Attack Types

Attacks exploiting memory corruption bugs, both temporal and spatial, can be divided into two categories based on the type of data they manipulate for exploitation.

Control Data Attacks

Control data attacks modify data that is at some later point loaded into the program's instruction pointer register and is therefore capable of directly affecting its control flow. Thus, modifying such control data allows an attacker to modify a program's behavior by diverting its control flow. Control data includes saved instruction pointers at the bottom of stack frames and function pointers, the latter of which can be located in any memory section.

Basic control data attacks were widely popularized by Elias Levy (Aleph One) in his 1996 article *Smashing the Stack for Fun and Profit* [26]. Here, he describes stack-based buffer overflow attacks overwriting the saved instruction pointer at the bottom of the stack frame, causing the program to jump to and execute user-supplied shellcode, i.e., a small piece of machine code constituting the attack payload. Although this technique was commonly usable for systems at that time, multiple countermeasures were introduced over the years, rendering this technique unusable on modern systems and causing the emergence of new exploitation techniques, which were again followed by new mitigations. Examples of newer exploitation techniques include return-to-libc and return-oriented programming (ROP) attacks.

While, in the early days of exploit development, the focus was primarily on stack-based OOB write vulnerabilities, exploitation of their heap-based counterparts gained traction in the early 2000s [27]. Heap-based OOB write vulnerabilities are typically more complex to exploit, not least due to the large variety of existing memory allocators, the characteristics of which are a crucial factor for assessing exploitability and crafting exploits.

Non-Control Data Attacks

Non-control data attacks are based on manipulating security-critical data in memory that does not directly alter the control flow of the exploited program. Chen *et al.* [28] provide several examples for such data that is commonly found in programs, including configuration data, decision-making data, file descriptors, and Remote Procedure Call (RPC) routine numbers. Non-control data attacks commonly require comparatively extensive knowledge of the program to be exploited, as the goal of exploitation is typically not as apparent as for control-data attacks, in which the aim is generally to manipulate a piece of data that is eventually loaded into the instruction pointer register. Instead, the path towards achieving non-control data exploitation and even the final goal are highly dependent on the individual program semantics [28].

A defining characteristic of non-control data attacks is that they do not breach control-flow integrity of the program. As such, the program will only perform valid transitions between basic execution blocks, even after exploitation. This allows non-control data attacks to evade detection by approaches based on ensuring control-flow integrity.

Although the general idea of non-control data attacks is straightforward and their feasibility has been known for a long time [28], they initially received relatively little attention, presumably as control data attacks are in many cases easier to perform. However, with the emergence of attack countermeasures in mainstream compilers, some of which are specifically tailored to prevent control data attacks [29], non-control data attacks have received increasing attention in recent years [30], [31].

2.3.2. Countermeasures

Over the last 30 years, several defenses against memory corruption attacks have been proposed, and some of them have been adopted by mainstream compilers. We provide an overview of the most widespread and successful ones, all of which are supported by most mainstream compilers.

We highlight each countermeasure's effectiveness against control data and non-control data attacks, confirming the intuition that most countermeasures used in practice are primarily aimed at and effective against control data attacks, while protection against non-control data attacks is often rather a convenient side effect.

Executable Space Protection Executable Space protection, also known as Data Execution Prevention (DEP) on Windows, is one of the earliest widely adopted exploitation countermeasures. It is based on the approach of marking memory pages as either executable or non-executable, causing the CPU to refuse the execution of instructions located on a non-executable page. On AMD64 CPUs, it is implemented through a dedicated NX bit in the page table, marking the page as executable [18]. It commonly follows a W^X (write xor execute) policy, dictating that a page can either be writable or executable, but not both.

The most substantial effect of Executable Space Protection is that it typically causes the stack to be non-executable. Thereby, exploitation approaches based on placing shellcode on the stack and triggering its execution become infeasible, thwarting simple attacks as described in *Smashing the Stack for Fun and Profit* [26].

Executable Space Protection is inherently only an effective countermeasure against control data attacks and does not influence non-control data attacks.

Address Space Layout Randomization Address Space Layout randomization (ASLR), first proposed by the PaX Team in 2001 [8], is a technique that randomizes the position of certain memory regions, thereby increasing the difficulty of creating reliable exploits that require knowledge of some entities' position in memory. Position-randomized regions are usually the stack, heap, and dynamically-linked shared libraries, as well as the memory-mapped segments of the executable file if it is compiled as position-independent code.

ASLR provides a substantial increase in resilience against exploitation, particularly on systems with an address space larger than 32 bits, and can, in some cases, effectively prevent the exploitation of vulnerabilities. However, it can be defeated by incorporating memory addresses leaked during program execution in the exploit, which can, for example, be leaked by OOB read vulnerabilities.

ASLR can increase resilience against both control data and non-control data attacks. However, it can be argued that ASLR tends to be more effective against control data attacks, as they frequently rely on the inclusion of memory addresses to which execution is to be redirected in their payloads.

Stack Canaries Canary-based protections such as stack canaries, first proposed in 1999 [7], are aimed at preventing the exploitation of classic buffer overflow vulnerabilities by placing a (typically random) canary value between the end of the buffer and security-critical data such as the saved instruction pointer. Before accessing the security-critical data, the canary value is compared against a copy. If they are not equal, e.g., due to the buffer having overflown, execution of the program is immediately terminated.

Canary-based protections are effective at increasing resilience against classical continuous buffer overflows but provide inadequate protection against non-continuous OOB writes that effectively 'jump over' the canary, as well as buffer underflows.

Although canary-based protections are in principle capable of thwarting both control data attacks and non-control data attacks, canary values are typically placed in locations where they protect control data, e.g., shortly before the bottom of the stack frame. As such, they are in practice primarily aimed at preventing control data attacks. Although it is possible to introduce canaries around any buffer, thus causing program termination on any continuous OOB write, this would introduce a non-negligible runtime performance overhead.

Control-Flow Integrity Control-flow integrity approaches generally aim to ensure that any control flow transition taken by a program is valid. As such, it is effectively a blanket term for any countermeasure against control data attacks, including most usages of canary-based protections. However, the term control flow integrity is most frequently used to refer to approaches that are explicitly designed to check the validity of control flow transitions at runtime. While this type of approach was first described by Abadi *et al.* in 2005 [32], numerous implementations of this are now available from various vendors, including purely software-based versions for the GCC and Clang compilers [29]. Approaches typically differ in their granularity, performance overhead, and compatibility with dynamically linked libraries.

A similarity of all approaches for ensuring control-flow integrity is that they are generally only aimed at thwarting control data attacks.

2.4. LLVM

LLVM is a compiler framework and toolchain originating from a research project at the University of Illinois in the early 2000s [33] that has grown to become the foundation for a large number of compilers, development tools, and research projects.

Components of the LLVM framework are primarily built around the LLVM intermediate representation (IR), which is a strongly-typed, low-level, assembly-like code representation in Static Single Assignment (SSA) form. It abstracts away from details of the target instruction set, such as physical registers and calling conventions, but is nevertheless capable of expressing higher-level concepts such as exception handling [33]. The LLVM IR, as of version 13 [34], consists of 65 different instructions and around 250 so-called intrinsic functions, the latter of which allow for performing common tasks such as logarithm computations and multi-byte memory copying. As such, the LLVM IR comprises substantially fewer instructions than ordinary CISC ISAs like AMD64.

Being in SSA form, the LLVM IR provides an unlimited number of typed virtual registers. Any used register is assigned a value exactly once, while it may be read multiple times in instructions following its assignment. Memory, on the other hand, is not represented in SSA form. Instead, it is generally accessed by dereferencing typed pointers via a small set of instructions and intrinsics, most notably the `load` and `store` instructions. LLVM contains two

types of identifiers: Global identifiers, which are prefixed with @, serve to identify functions and global variables, while local identifiers, prefixed with %, are used for virtual registers and types.

The LLVM IR exists in three different representations, all of which are equivalent: A human-readable, assembly-like textual representation, a bitcode representation, and an in-memory representation for just-in-time compilation.

The LLVM components involved in the compilation process are strictly separated, as displayed in Figure 2.5. The *frontends* are responsible for translating the source code of a high-level language such as C, C++, Rust, or Swift to the IR. Afterwards, the generated IR is typically subjected to a number of so-called *passes*, which can modify the IR and are aimed at performing various optimizations and analyses on it. Finally, the optimized IR is translated to machine code by the *backend* for the target architecture. This process is typically applied to each module of the program in isolation, resulting in the compiler generating an object file with machine code for each module. These object files are eventually combined into an executable or shared library by the linker.

A consequence of this approach is the inability to perform effective intraprocedural optimization with a program-level scope, which is remarkably difficult to realize for machine code. LLVM mitigates this limitation by facilitating link-time optimization (LTO) through the emission of LLVM IR instead of machine code by the compiler. This allows the linker to perform intraprocedural whole-program optimization on the LLVM IR, using the rich semantic information it contains. The task of generating machine code from the LLVM IR is hereby delegated to the linker.

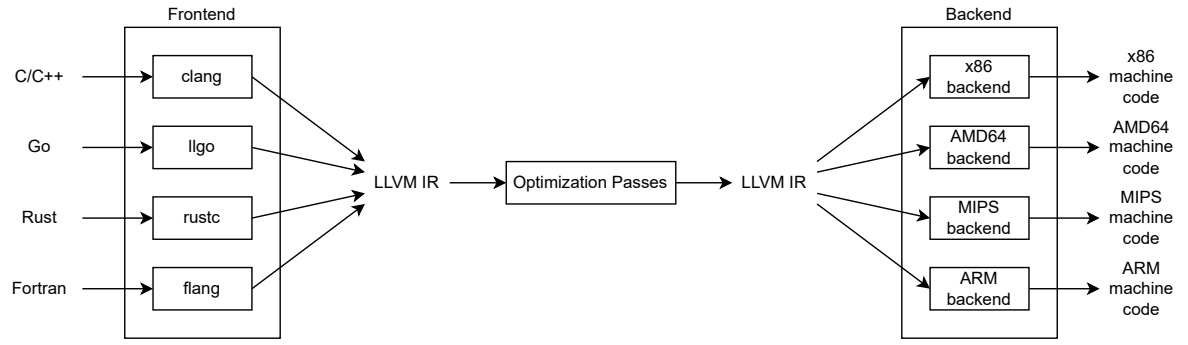


Figure 2.5: Compilation from source code to machine code with LLVM

The strengths of LLVM that caused it to become as widely used and popular as it is today are apparent. Creating a compiler for a new programming language with LLVM only requires the creation of a corresponding frontend - optimization passes and the backends operate on the IR and can therefore be reused. Analogously, supporting a new target architecture only requires the creation of a corresponding backend. Furthermore, the development of custom analysis and optimization passes is eased by the well-defined IR [34].

2.5. S2E

S2E is a platform for in-vivo multi-path analysis of software systems, capable of applying selective symbolic execution to large programs in an emulated environment [35]. It is based on the QEMU emulation software [36], the KLEE symbolic execution engine [37], and LLVM [33]. S2E uses a modified Linux kernel to execute binaries within a complete software stack and provides an API for performing fine-grained analysis.

Like QEMU, S2E translates guest instructions in chunks of so-called Translation Blocks (TLBs) for execution, each of which ends in a control flow change. Contrary to QEMU, however, not all TLBs are translated to native instructions and executed. Instead, TLBs that touch symbolic data are lifted to the LLVM IR and concolically executed by KLEE. This approach allows for a substantial execution speedup of code that does not touch any symbolic data while maintaining the ability to execute concolically. It is facilitated by a shared representation of memory, CPU state, and IO devices that is used and updated regardless of whether a TLB is executed natively or in KLEE [38].

S2E is designed to perform on-line concolic execution. As such, it forks the state of the program under test upon encountering a branch on a symbolic value, allowing for continuing its execution as soon as the resources to do so are available.

A significant strength of S2E is its ability to switch between execution modes seamlessly. For example, a binary might be executed in concolic mode, causing S2E to repeatedly create forks of the program state to explore until the state reaches a location in the program for which exploration is not desired, e.g., a call to a library function. At this point, S2E can suspend forking as long as library code is executed and continue forking upon returning from the library function call. This can be an effective countermeasure against state explosion, which is one of the major drawbacks of symbolic execution in general.

2.6. DWARF

The DWARF debugging information format [39] is a standardized format for embedding debug information into object files and constitutes the de facto standard for including debugging information in ELF files. It can be generated by virtually any compiler generating ELF files and supports various programming languages, including C and C++.

The core of a program's DWARF debugging information, located in the `.debug_info` section of the ELF file, is structured using so-called Debug Information Entries (DIEs), which are organized in a tree-like hierarchical fashion, and each represent an entity such as a compilation unit, a function, or a variable. Each DIE contains zero or more attributes that contain information, such as the source code-defined name of the function it represents or a reference to the DIE that holds information on the type of the variable it represents. The totality of DIEs provides a detailed description of the program's structure and can be used to learn a variety of source code-level information about the program at run time, such as the structure of data types and the position of variables in memory.

An example for DIEs, generated by compiling the function in Figure 2.6 without optimizations and with full debug information, is presented in Figure 2.7. In the example, the uppermost DIE represents the function itself, and its attributes describe the function's position in the ELF file, the frame base register (`rbp`), return type, and further source code-level details. The subprogram DIE representing the function again has two children - one for the function argument and one for the local variable. As the program was compiled without optimizations, both are located on the stack. Their positions relative to the frame base register are specified in their DIE's attributes, along with their name, type, and source code location.

A further type of DWARF debugging information, found in the `.debug_line` section, is line number information. It contains a mapping of instruction addresses to corresponding source code locations, specified by the filename, line, and column.

```c
double getEqlTriangleArea(double side_len) {
  double x = sqrt(3) / 4;
  return x * side_len * side_len;
}
```

Figure 2.6: Simple C function

```
DW_TAG_subprogram
  DW_AT_low_pc  (0x0000000000401160)
  DW_AT_high_pc (0x00000000004011b4)
  DW_AT_frame_base  (DW_OP_reg6 RBP)
  DW_AT_name  ("getEqlTriangleArea")
  DW_AT_decl_file ("/home/linus/demo/demo.c")
  DW_AT_decl_line (15)
  DW_AT_prototyped  (true)
  DW_AT_type  (0x00000171 "double")
  DW_AT_external  (true)

  DW_TAG_formal_parameter
    DW_AT_location  (DW_OP_fbreg -8)
    DW_AT_name  ("side_len")
    DW_AT_decl_file ("/home/linus/demo/demo.c")
    DW_AT_decl_line (15)
    DW_AT_type  (0x00000171 "double")

  DW_TAG_variable
    DW_AT_location  (DW_OP_fbreg -16)
    DW_AT_name  ("x")
    DW_AT_decl_file ("/home/linus/demo/demo.c")
    DW_AT_decl_line (16)
    DW_AT_type  (0x00000171 "double")
```

Figure 2.7: DWARF DIEs representing the simple C function

3

Related Work

With memory unsafe programming languages like C having dominated the software world for decades, research into ways to detect, analyze and mitigate memory access errors like OOB writes has been taking place for more than 30 years. On a high level, existing OOB write detection approaches generally fall into two categories.

Memory sanitizers commonly serve to facilitate the detection of memory safety violations at runtime in a stand-alone fashion, relying on instrumentation inserted into the program under test to perform monitoring. As their most important use case is the identification of memory safety violations at the development or testing stage, e.g., in the context of fuzzing, a low performance and memory overhead is commonly an important goal in their design. Nevertheless, existing memory sanitizers still introduce substantial overhead, making them unfit for use in production settings, i.e., for exploit mitigation [40]. Furthermore, memory sanitizers are typically built with the primary goal in mind to merely detect the existence of a memory safety violation and, in some cases, provide information to aid a developer in fixing it. However, identifying its concrete security implications is commonly not an important goal.

Pipelines, e.g., for crash triaging or automatic exploit generation that rely on detecting OOB write vulnerabilities, are another use-case for OOB write detection approaches. Here, functionality requirements typically differ from those for memory sanitizers, with overhead commonly being less of a concern.

We will further provide an overview of existing techniques for detecting OOB writes, followed by an overview of existing implementations of memory sanitizers and an overview of pipelines implementing some type of OOB write detection.

3.1. Technique Overview

Existing OOB write detection approaches differ in several fundamental aspects that affect their capabilities, usability, and performance. We hereafter provide an overview of the most important aspects.

3.1.1. Instrumentation

One of the fundamental aspects in which OOB write detection approaches differ is how they monitor the program under test during execution.

Approaches leveraging compile-time instrumentation (CTI) [23], [41]–[44] insert their checking logic into the program's source code or intermediate representation during compilation. Such approaches benefit from the availability of detailed, high-level information on memory object boundaries at this stage. Clearly, such approaches are only suitable for scenarios where the source code or intermediate representation is available. Furthermore, the lack of control over the resulting assembly code usually makes these approaches fairly invasive by modifying not only the code but also the program's memory structure during execution, particularly the stack layout. While such modifications are usually unproblematic for sanitization, they can make the approach unusable for vulnerability analysis and scenarios in which OOB writes are not only to be detected but also investigated in more detail, such as in automatic exploit generation.

Approaches leveraging static binary instrumentation (SBI) [45] or dynamic binary instrumentation (DBI) [22], [46], on the other hand, operate solely on compiled code. As such, they do not require access to source code or intermediate representation and are therefore suitable for use in a greater variety of scenarios. As their checking logic is inserted directly into the program's assembly, they allow for a far more fine-grained control of modifications to the program and, as such, can be made noninvasive in terms of memory structure modification. However, due to the lack of high-level information on memory object boundaries at the assembly level, such approaches rely on heuristics to

determine object boundaries against which memory accesses are checked for legality. This not only prevents these approaches from making strong spatial detection guarantees but also causes a substantial increase in complexity.

3.1.2. Detection Approach

A further aspect by which different OOB write detection approaches can be distinguished is their technique for differentiating between legitimate memory writes and OOB writes. On a high level, these techniques can be divided into two categories: identity-based approaches and tripwire approaches.

Identity-based OOB write detection

Identity-based approaches detect OOB writes by comparing the memory object expected to be accessed according to the semantics intended by the programmer with the memory object that is accessed and raising an alert if they do not match.

For instructions that always write to a fixed memory object, this intended destination memory object can be straightforwardly determined by static analysis. With the destination object being known, its bounds can either be determined from the program if they are fixed at compile time or obtained from within the function scope if they are determined dynamically, e.g., in the case of a variable-length array. This allows for easily instrumenting the program to perform bounds checking before write operations.

For instructions that write to a location determined by a pointer, on the other hand, the intended destination object depends on the source of the dereferenced pointer and, therefore, on the control flow of the program. As such, OOB write detection for this type of writes is far from trivial. Maintaining a mapping between pointers and intended destination objects or their bounds is the core contribution of most works on identity-checking spatial memory error detection approaches.
A major distinction between different approaches for detecting such errors is the entity with which bounds are associated, which can either be the pointer itself or the pointee object.

Pointer-based approaches [41], [43], [44] associate bounds with individual pointers. This requires augmenting pointers with metadata, usually by embedding this information directly into the pointer, as described below. Metadata is embedded into pointers by instrumented pointer creation sites and propagated to derived pointers when performing pointer arithmetic. Upon dereference, it is checked that the pointer is still in bounds, according to its metadata. An important advantage of pointer-based metadata management is the ability to perform intra-object bounds checking, as metadata is managed for each pointer separately. This allows two pointers to the same address to have different metadata associated with them.

Object-based approaches [47] associate metadata only with memory objects. Instead of tagging pointers upon creation and checking their value and tag before dereferencing, they commonly monitor program locations where arithmetic operations are performed on pointers. By instrumenting such locations and maintaining a collection of existing memory objects and their bounds, the pointee object of a pointer can be determined before and after the arithmetic operation. If they are different, the pointer has gone out of bounds. When this occurs, the pointer is usually modified to point to an address or object that causes the OOB write to be reported upon dereferencing.
A clear drawback of this approach is the inability to monitor for intra-object OOB writes, as the address range of a composite datatype will inevitably overlap with the address space of its fields, and the exact pointee can therefore not be determined from the pointer alone.
Consider for example a structure with the definition `struct User {char name[64]; int balance;}`. A pointer to the beginning of the structure points to the same address as a pointer to the field `name`. To avoid false positives when accessing the field `balance` from a pointer to the beginning of the structure, one must consider this pointer to be for the entire structure, not just for the first field. However, this makes it impossible to detect an overflow of the first field.

Augmenting Pointers Most identity-based OOB write detection approaches rely on associating metadata with pointers by changing their internal representation. Two different approaches to this are commonly found.

Pointer tagging Pointer tagging [48] makes use of the fact that the most significant byte in pointers on some 64-bit systems such as AMD64 remain unused for memory addressing [19]. This byte is used to encode a tag into the pointer that is shared between a memory region and any pointer pointing to it. Upon every dereference, the pointer's tag is compared to the tag of the memory region. However, due to the restricted space for storing the tag in pointers, tags are usually created randomly, causing probabilistic matching between pointers and memory regions with a certain chance for false negatives [42].

If implemented purely in software, pointer tagging requires instrumenting any deference to perform bounds checking, as well as bookkeeping of tags and removing the tag from the pointer before dereferencing it. As such, non-instrumented code such as libraries would cause a segmentation fault upon dereferencing a tagged pointer.

Pointer tagging with partial hardware assistance, on the other hand, uses special CPU instructions available for certain ISAs to efficiently strip tags from pointers. While this still requires instrumenting all code, such instrumentation can be considered less invasive and easier to perform reliably.

Pointer tagging with full hardware assistance entirely delegates tag creation, management, and checking to the hardware. However, runtime libraries and instrumentation are still required for programs to utilize this, and few ISAs currently support pointer tagging.

Fat pointers Fat pointers replace normal-sized pointers with structures containing at least the pointer value, pointee base address, and pointee size. While this allows for associating a large amount of data with each pointer, it requires rather invasive program modifications. The calling convention needs to be modified to support fat pointers and pointer creation sites, pointer dereference sites, and pointer arithmetic sites need to be instrumented.

Neither pointer tagging nor fat pointers are directly compatible with non-instrumented code such as external libraries and require stripping metadata from the pointer beforehand.

Tripwire-based OOB write detection

Tripwire-based approaches [22], [23] circumvent the challenge of determining the memory object expected to be accessed according to the semantics intended by the programmer. This is done by inserting new memory regions between memory objects, usually referred to as (poisoned) red zones. A check before or after each memory access asserts that no such red zone is accessed, reporting any access as OOB. The insertion of red zones necessitates bookkeeping to keep track of the red zone locations. For this, shadow memory, i.e., a separate memory region for efficiently recording information about the application's memory through a mapping function, is commonly used.

An inherent shortcoming of this approach is that non-continuous OOB writes that "jump over" the red zone remain undetected. As such, tripwire-based OOB write detection cannot make strong spatial detection guarantees. Furthermore, tripwire-based approaches are invasive by design. Heavy modification of the program's memory layout is inevitable, making it challenging to transfer insights into the spatial properties of OOB write vulnerabilities to the non-instrumented program.

3.1.3. Monitored Regions

A further distinction between OOB write detection approaches are the program sections for which OOB writes are detected. In programs linked with a modern linker using standard settings, the important writable regions are the global sections (`.bss` and `.data`), as well as heap and stack. As such, they, or a subset of them, are the focus of virtually all OOB write detection approaches. While OOB write detection in globals and on the stack require either modification of the section's memory layout through instrumentation [23] or excavation of metadata from symbol table or debugging information [46], coarse-grained OOB write detection on the heap can be achieved by monitoring or hooking into calls to dynamic memory allocation functions such as `malloc` [22], [23], [41]–[44].

3.2. Memory Sanitizers

Sanitizers are instrumentation-based tools for detecting certain types of anomalous behavior in programs at runtime. Such anomalous behavior comprises spatial memory violations such as out-of-bounds writes and reads, temporal memory violations such as use-after-free conditions and uninitialized reads, race conditions, as well as other undefined behavior such as integer overflows. However, given the goal of our work, we will hereafter focus exclusively on those sanitizers capable of detecting out-of-bounds writes. As dozens of sanitizers with such capabilities exist, we cannot provide an exhaustive overview of them and instead present only the most relevant spatial memory sanitizers. A tabular overview of their important characteristics is presented in Table 3.1.

AddressSanitizer (ASan) [23] is presently the most widely adopted sanitizer and is included in the compilers GCC and Clang [40]. It is based on a tripwire approach and performs CTI to insert red zones around stack and global objects, as well as checks before any memory access. A runtime library is used to intercept library calls for dynamic memory management, thereby inserting red zones around allocated memory chunks on the heap. Red zones are kept track of through shadow memory with a granularity of one byte, which allows for tracking 8 bytes of normal memory with a single byte of shadow memory. This, among others, causes ASan to incur a comparatively small average performance overhead of 73% at the cost of 340% memory overhead.

ASan can detect spatial memory violations on the heap, stack, and in the global variables, as well as UAF conditions. While it can detect any continuous inter-object OOB write in the regions mentioned above, non-continuous OOB writes that effectively jump over red zones into neighboring objects generally remain undetected. The same typically applies to intra-object OOB writes, which can be detected when the source code is extended with ASan-specific annotations but remain undetected otherwise.

Hardware-assisted AddressSanitizer (HWAsan) follows an essentially different approach despite the naming similarity. HWAsan implements object-based OOB detection based on pointer tagging by utilizing the top-byte-

	ASan [23]	HWAsan [42]	Memcheck [22]	SGcheck [46]	SoftBound [41]	Delta Pointers [44]	Intel MPX [43]	BinArmor [45]	PAriCheck [47]
Detection of OOB writes in globals	✓	✓	×	✓	✓	✓	✓	✓	✓
Detection of OOB writes on stack	✓	✓	×	✓	✓	✓	✓	✓	✓
Detection of OOB writes on heap	✓	✓	✓	×	✓	✓	✓	✓	✓
Strong spatial guarantees	×	×	×	×	✓	×	✓	×	×
Intra-object OOB write detection	×	×	×	×	(✓)	×	✓	✓	×
Instrumentation type	CTI	CTI	DBI	DBI	CTI	CTI	CTI	SBI	CTI
No invasive memory layout modification	×	×	✓	✓	×	×	×	×	×
No need for hardware support	✓	×	✓	✓	✓	✓	×	✓	✓
Compatible with external code	✓	×	✓	✓	×	✓	✓(?)	✓(?)	✓
Detection approach	TW	PB	TW	HR	PB	PB	PB	PB	OB
Performance overhead	L	L	H	?	L	L	M	M	L
Memory overhead	H	L	?	?	L	L	M	?	L

Strong spatial guarantees: Ability to non-probabilistically detect non-continuous and underflowing OOB writes.
TW: tripwire, *OB*: object-based, *HR*: heuristics, *PB*: pointer-based
L: 0x-1x overhead, *M*: 1x-3x overhead, *H*: 3x+ overhead, *?*: no data

Table 3.1: Overview of important characteristics of existing spatial memory sanitizers

ignore instruction of the ARM64 ISA to strip tags from pointers efficiently. Upon dereference, the pointer tag is compared to the pointee tag. This removes the necessity of sacrificing memory space for red zones and allows for reducing the size of the shadow memory, which is now required for storing the tag of shadowed bytes. Thus, HWAsan reduces memory overhead to less than 50% [42], compared to the unmodified program. A further consequence is the ability to detect non-continuous OOB accesses that ASan might miss. However, with the tag bit-size TS being very limited, duplicates are possible, and a comparison of two tags results in a false negative with a probability of $\frac{1}{2^{TS}}$.

Kernel address-sanitizer [49] (KASAN) can be considered the kernel space variant of ASan, tailored for the Linux kernel. Besides a purely software-based approach as used by ASan, it also implements a partially hardware-assisted mode similar to HWAsan. Furthermore, it implements a mode with full hardware assistance that uses the memory tagging extension of the ARM64 ISA to perform not only stripping of tags but also their creation, bookkeeping, and checking.

SoftBound [41] performs CTI by means of multiple LLVM passes to achieve strong spatial detection guarantees. Although it follows a pointer-based approach, it does not modify the representation of pointers. Instead, it ensures that any pointer is accompanied by two variables containing its base address and size while traveling through the program. As such, it modifies function interfaces to pass base address and size along with every pointer passed to a function and replaces returned pointers with structures consisting of pointer and its metadata, passed by value. A dedicated metadata lookup table is used to maintain metadata for pointers written to memory, in which the memory address at which the pointer is stored serves as the key.
Although SoftBound is described as capable of detecting intra-object OOB writes in the corresponding publication, we found that the publicly available prototype implementation does not provide intra-object OOB write detection. Furthermore, SoftBound is not directly compatible with non-instrumented code such as libraries due to the heavy modification of function interfaces. Thus, wrappers must be used to achieve compatibility with non-instrumented functions. Besides providing compatibility with non-instrumented functions, such wrappers can also be used to perform bounds checking of memory accesses by standard library functions.

Memcheck [22], which is part of Valgrind [50], performs dynamic binary instrumentation to capture dynamic memory management function calls and redirect them to its implementation of the corresponding function. Besides facilitating the detection of some temporal memory violations and memory leaks, it inserts red zones around allocated dynamic memory regions, thereby detecting continuous OOB writes with a tripwire approach. However, as the main focus of Memcheck is the detection of undefined value errors and OOB access detection is merely a secondary feature, Memcheck does not make any attempts to detect OOB accesses on the stack or in global variables.

Complementary to Memcheck, the experimental Valgrind tool SGcheck [46] leverages debug information of binaries to perform OOB access detection for arrays on the stack and in the global sections. It attempts to overcome the lack of information about the intended target of a memory-accessing instruction by using a simple heuristic based on the assumption that if a particular instruction accesses an array once, it will only access that particular array in the future. As such, SGcheck follows neither an identity-based nor a tripwire-based approach. However, as its heuristic effectively assumes that a given instruction will only ever write to a single variable, false positives and negatives are likely to occur.

Delta Pointers [44] performs compile-time instrumentation to detect overflows with a focus on minimizing performance overhead by refraining from doing branching checks. Pointers are tagged with two pieces of information contained in the n most significant bits. The most significant bit serves as an overflow indicator and causes the memory management unit to report a segmentation fault upon dereferencing an OOB pointer, thus eliminating the need for explicit bounds checks. The remaining $n-1$ most significant bits are used to encode the pointer's current distance from the corresponding object's upper bound. Arithmetic operations on pointers are instrumented to modify the encoded distance and, if necessary, the overflow bit accordingly. Before dereferencing a pointer, all tagging bits except the overflow bit are stripped.
While Delta Pointers cause a remarkably small performance overhead of 35% and no memory overhead, as metadata is exclusively encoded in pointers, the main limitation is the inability to detect underflows.

Intel Memory Protection Extensions (MPX) [43] is an extension to the AMD64 instruction set architecture (ISA) that, in combination with compile-time instrumentation and two runtime libraries, provides functionality for achieving complete spatial memory safety. However, as a consequence of multiple design flaws and poor performance despite its implementation in hardware, MPX was deprecated in 2019 and is not supported anymore by modern compilers and Intel CPUs.

BinArmor [45] performs static binary instrumentation to detect OOB writes in global sections, on the heap, and on the stack using a pointer-based approach similar to SoftBound. It performs data structure recovery through dynamic analysis to compensate for the lack of debug information and symbol tables frequently encountered in closed-source binaries. However, despite its ability to perform intra-object OOB write detection, it cannot provide strong spatial detection guarantees due to the heuristic-based approach to data structure recovery, the accuracy of which depends on the achieved code coverage.

PAriCheck [47] is an object-based approach using compile-time instrumentation to insert checks around pointer arithmetic sites. While one consequence of its object-based approach is its incapability of detecting intra-object OOB writes, it also incurs a remarkably small performance and memory overhead.

3.3. Pipeline Integrations

A further important use case for OOB write detection mechanisms are pipelines integrated into which they facilitate automatic exploit generation or bug triaging.

KOOBE [51] is an automatic exploit generation pipeline for heap-based OOB write vulnerabilities in the Linux kernel. It is designed in a modular fashion, decoupling the capability exploration of a vulnerability from the exploit synthesis. The starting point for KOOBE is a PoC input in the form of a system call with arguments that cause an OOB write within the kernel's dynamic memory. KASAN is then used to identify the source code locations at which OOB writes are taking place. After this, these source code locations are mapped to instruction addresses using static analysis. These instruction addresses and the PoC are then passed on to S2E, which is used for fine-grained analysis of the OOB write capabilities, particularly in terms of writable addresses.
This two-stage approach using KASAN and S2E for OOB write detection combines the benefits of both techniques: the comparatively low performance overhead of KASAN improves the overall runtime performance by directing the fine-grained analysis with S2E towards locations where OOB writes are taking place with certainty, while the analysis with S2E is capable of catching specific OOB writes that would remain undetected by KASAN due to the weak spatial detection guarantees incurred due to its tripwire approach. Furthermore, the symbolic constraints generated by S2E can be used to reason about vulnerability capabilities without constructing concrete inputs that exhibit the capability. However, by using KASAN for initial detection, OOB writes that already jump over KASAN's red zone in the PoC input will remain fully undetected and not be subject to analysis with S2E.

BORG [52] is a testing tool focused on discovering buffer overreads in binaries and therefore implements functionality for detecting OOB reads, which, despite having entirely different security implications, can be detected and reasoned about similarly to OOB writes. As the tool operates solely on binary executables, an important distinction from other techniques for detecting OOB accesses is that it cannot leverage information about object boundaries from the source code. Instead, it collects per-state buffer access profiles during a preliminary analysis stage in which a given set of test inputs is executed. For each state, defined as the current call stack in combination with the instruction to be executed, the profile consists of the buffers observed to be accessed at this state. During the testing stage, memory read accesses are checked for legitimacy by checking whether the location to be read is part of a buffer and whether this buffer is contained in the corresponding profile. If the location to be read is not part

of any known buffer, an OOB read is deemed likely. On the other hand, if it is part of a known buffer, but that buffer is not contained in the state's profile, the access is considered suspicious. As such, it follows a similar heuristic approach as SGcheck.
The downsides of this approach are that no OOB reads in any code not covered by a test input can be detected, as well as the implicit assumption that test inputs themselves do not trigger any OOB reads.

The authors of Revery [53], a pipeline for automatic exploration and exploit generation of heap-based vulnerabilities, acknowledge the impracticality of using memory-manipulating tools like Valgrind, ASan, and SoftBound for automatic exploit generation. Instead, they follow a pointer tracking approach using memory tagging, which is implemented purely in software and uses shadow memory for metadata management. However, besides UAF conditions, their implementation can only detect inter-object OOB writes.

Numerous pipelines for automatic generation of control flow-diverting exploits circumvent the challenge of performing detailed OOB write detection by simply monitoring for indicators of the instruction pointer being directly affected by user-supplied input. In practice, this corresponds to the instruction pointer having become symbolic [14] or tainted [13] in the case of symbolic execution and taint analysis, respectively.

4

Motivation

In this chapter, we first describe our motivation for designing a non-invasive OOB write detection approach by highlighting the shortcomings of existing OOB write detection approaches for distilling vulnerability characteristics in binaries that follows from their invasive behavior. Afterwards, we position our approach, described in detail in Chapter 5, relative to existing approaches.

4.1. Issues of Existing Approaches

Despite the abundance of approaches and tools for detecting OOB writes, of which the most important ones are described in Chapter 3, none of these off-the-shelf solutions are suitable for our use-case of distilling capabilities from discovered OOB writes. To the best of our knowledge, all publicly available approaches and tools suffer from one or more of the following issues that make their use impractical for our use case of identifying the effective source code-level consequences of an out-of-bounds write caused by a given input.

Inability to detect intra-object Out-of-bounds writes Many approaches are incapable of detecting intra-object OOB writes within composite objects such as structures. However, intra-object OOB writes are well capable of inducing security issues. Therefore, their inclusion in a vulnerability's capability profile is essential, and our detection approach must be able to identify them just like inter-object OOB writes. As an example of the possible danger of intra-object OOB writes, consider the structure shown in Figure 4.1. Here, an overflow of the `username` array could corrupt the `isAdmin` flag stored after it in the structure, potentially facilitating a non-control data attack.

Required hardware support Some approaches [42], [43], [48] rely on extensions to the ISA of the CPU to perform OOB write detection. While such ISA extensions are available for SPARC [48], ARM64 [42], and some historic Intel AMD64 CPUs [43], they are not included in the ISA of any recent AMD64 CPUs. Therefore, we cannot rely on any type of hardware support.

Invasive modification of the program Almost all existing OOB write detection approaches we reviewed cause the program's memory layout to be modified due to their instrumentation.

To understand the problems for our use case following from this, we first need to look at how instrumentation causes the memory layout of the program under test to be modified. In general, such modifications performed by OOB write detection approaches usually fall into one or more of the following categories.

1. Introduction of poisoned red zones around objects on the stack, heap, and in global sections
2. Introduction of new memory regions to store metadata, e.g., as shadow memory
3. Direct and indirect modification of stack frames caused by storing metadata and performing checks

Modifications of the first category are often predictable due to the usage of fixed-size red zones, potentially supplemented by alignment padding, and therefore only pose a moderate issue for our use case. Modifications of the second category are even less of an issue, as they commonly take place far off the memory regions used by the program under test itself and are limited to those regions. However, modifications of the third type are highly problematic for our use case.

We illustrate modifications of the third type by the example of the function presented in Figure 4.1. This function performs simple password checking functionality and, despite not being vulnerable, can be considered a typical location for mounting both control data and non-control data attacks. We present a visual mock-up of the function's

stack frame when compiled with default settings, using ASan [23], and using SoftBound[1] [41] in Figure 4.2. Here, it is clearly visible that both ASan and SoftBound heavily modify the stack frame, causing its size to grow by a factor of more than two and three, respectively. A further interesting observation is that, for the non-instrumented function, the variable `pwOk` is not saved to memory and is kept exclusively in a register, while it is stored in memory for both instrumented versions of the function.

While the source code object's positioning relative to each other can be seen to follow a certain pattern, with ASan introducing 32-byte red zones around the objects after padding them to be 16-byte aligned and reversing their order, and SoftBound placing all objects approximately adjacent to each other, their placement within the overall stack frame is not as predictable. This is caused both by the heavy usage of the stack for storing bounds metadata and the occupation of registers by the checking logic, which causes the number of general-purpose registers available for the remaining function logic to decrease and thereby the occurrence of register spilling to the stack to increase, using up additional stack space.

These modifications to the stack are problematic for two reasons.
Firstly, a modified layout of the stack frame in which an OOB write occurs makes it highly challenging to reliably determine the objects that would be affected by this OOB write in the non-instrumented program. Especially the effects of register spilling and compiler optimizations on the stack frame layout are difficult to gauge. This is particularly problematic if no ground-truth information about the non-instrumented program's memory layout is available. However, even with access to the ground-truth memory layout of the non-instrumented program, which can be obtained by compiling it without the instrumentation, mapping the affected memory objects in the instrumented binary to those in the non-instrumented binary to determine the affected objects would remain a complex and challenging task, for which it is questionable whether it can be performed soundly.
Secondly, modification of the stack is accompanied by the risk of altering the program's control flow after anomalous behavior such as an OOB write. This can cause the instrumented program to behave in ways that are impossible for the non-instrumented program and deliver incorrect results about the effects of OOB writes. Such behavior might be facilitated by our need to continue executing the program even after an OOB write has been detected, as we strive to discover its full effects. As an example, take a buffer that is susceptible to an overflow and followed by a function pointer in memory. If the buffer overflows into the function pointer in the non-instrumented program, the result would most likely be a segmentation fault upon dereferencing the function pointer by calling it. In the instrumented program, however, additional space between the buffer and the function pointer might not cause the function pointer to be overwritten by the overflow, thus allowing for execution to continue and arrive in states that can never be reached in the non-instrumented program.

```
1  struct userEntry {
2    char username[32];
3    int id;
4    bool isAdmin;
5  };
6
7  void login(struct userEntry* userPtr) {
8    struct userEntry user;
9    char realPw[32];
10   char tryPw[32];
11   bool pwOk;
12   user = *userPtr;
13   getUserPassword(&user, realPw);
14   puts("Enter your password: ");
15   fgets(tryPw, 32, stdin);
16   pwOk = ! strcmp(realPw, tryPw);
17   if (pwOk) {
18     setUserLoggedIn(&user);
19   }
20 }
```

Figure 4.1: Example function to demonstrate stack layout manipulation

[1] As SoftBound is not fully functional for Clang > 3.8.0 and ASan in Clang < 3.9.0 contains a bug that makes it incompatible with recent versions of `glibc`, Clang 4.0.0 was used for compilation without sanitization and with ASan, while Clang 3.8.0 was used for compiling with SoftBound.

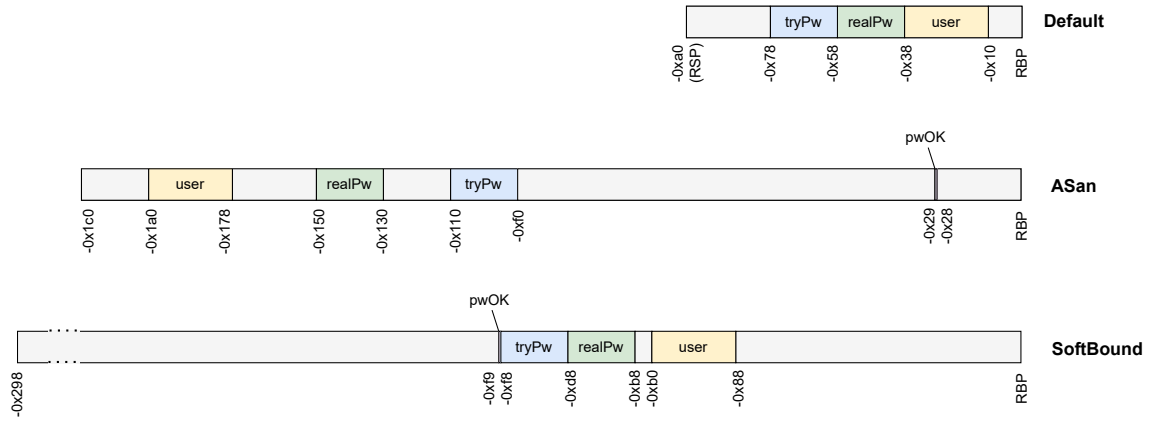


Figure 4.2: Stack layout of function in Figure 4.2 when compiled without instrumentation, with ASan, and with SoftBound

Instability after Out-of-bounds write Most OOB write detection tools are designed to terminate the program's execution immediately after detecting an OOB write. While, for most approaches, this behavior can be circumvented by applying some modifications to the approach, doing so might cause the program to become unstable and crash or result in other execution paths that are impossible for the non-instrumented program, similar to the invasiveness issue discussed above.

For example, consider the case in which an OOB write detection approach uses the stack to store metadata such as bounds information, which is done by most approaches either directly or indirectly through registers being spilled to the stack. Now, suppose an OOB write occurs that overwrites this stored metadata. In that case, the consequences are largely unpredictable, ranging from a continued ordinary execution to the occurrence of false positives or false negatives or a crash of the entire program.

As we wish to reliably detect more than just the first OOB write occurring during the execution of a program, particularly if the OOB write occurs within a loop that writes to memory in every iteration, this limitation makes most existing OOB write detection approaches unusable for our use case.

4.2. Positioning Our Approach

Motivated by the limitations of existing approaches just discussed, which make them unfit for the use case of distilling the capabilities of an OOB write in terms of the affected memory objects, we propose our own approach for OOB write detection in Chapter 5.

We can position our approach to the existing approaches discussed in Chapter 3 as presented in Table 4.1. Our approach is a pointer-based OOB write detection approach that is able to detect spatial memory violations on the stack and in the global program sections in a deterministic fashion. It is capable of providing strong spatial detection guarantees that even extend to external library code and manages to detect both inter- and intra-object OOB writes. By not interfering with the compilation in any way and leaving the resulting machine code untouched, it is entirely non-invasive and does not rely on special hardware support. As such, we can be certain that insights about the spatial effects of OOB write vulnerabilities in the examined binary also hold for the program when it is not monitored by our approach. Furthermore, we can be certain that any path taken through the program during our analysis is also a possible path during normal execution of the program and vice versa.

However, as the implementation of our approach relies on emulating the entire system in which the program under test runs, it incurs a high performance and memory overhead.

	ASan [23]	HWAsan [42]	Memcheck [22]	SGcheck [46]	SoftBound [41]	Delta Pointers [44]	Intel MPX [43]	BinArmor [45]	PAriCheck [47]	Our Approach
Detection of OOB writes in globals	✓	✓	×	✓	✓	✓	✓	✓	✓	✓
Detection of OOB writes on stack	✓	✓	×	✓	✓	✓	✓	✓	✓	✓
Detection of OOB writes on heap	✓	✓	✓	×	✓	✓	✓	✓	✓	×
Strong spatial guarantees	×	×	×	×	✓	×	✓	×	×	✓
Intra-object OOB write detection	×	×	×	×	(✓)	×	✓	✓	×	✓
Instrumentation type	CTI	CTI	DBI	DBI	CTI	CTI	CTI	SBI	CTI	n.a.
No need for hardware support	✓	×	✓	✓	✓	✓	×	✓	✓	✓
No invasive memory layout modification	×	×	✓	✓	×	×	×	×	×	✓
Compatible with external code	✓	×	✓	✓	×	✓	✓(?)	✓(?)	✓	✓
Detection approach	TW	PB	TW	HR	PB	PB	PB	PB	OB	PB
Performance overhead	L	L	H	?	L	L	M	M	L	H
Memory overhead	H	L	?	?	L	L	M	?	L	H

Strong spatial guarantees: Ability to non-probabilistically detect non-continuous and underflowing OOB writes.
TW: tripwire, *OB*: object-based, *HR*: heuristics, *PB*: pointer-based
L: 0x-1x overhead, *M*: 1x-3x overhead, *H*: 3x+ overhead, *?*: no data

Table 4.1: Important characteristics of existing memory sanitizers, compared to our proposed approach

5 Design

This chapter provides a detailed description of our non-invasive OOB write detection approach, along with the conceptual framework that facilitates it by making source code-level information available to the low-level detection logic.
First, we define a number of general concepts the remaining chapter relies on, along with our scope for detection. After that, we provide a high-level description of our approach, isolating the core challenges in our design. Our solutions for these core challenges are then addressed in the remaining sections of this chapter.

5.1. Defining ”Out-Of-Bounds”

A detailed description of which types of memory accesses are considered out-of-bounds is frequently omitted in existing publications on memory sanitizers, although it can be seen as a crucial part in the design of a detection approach. A common implicit assumption appears to be that any memory access described as legal by the C standard is in-bounds, while all types of out-of-bounds accesses are specified as undefined behavior in the standard. However, the C11 standard [4] fails to adequately specify whether certain ways of accessing memory result in defined or undefined behavior, leaving room for interpretation. An example of this is accessing a member of a structure with a pointer to another member of the same structure. To provide a concrete example, it is not clearly defined whether, for the structure `struct {int a, int b, int c} s`, accessing the member `c` using `(&s.b)[1]` constitutes defined behavior. Therefore, we provide a description of which memory accesses we consider out-of-bounds. As our approach aims to detect OOB writes at the assembly level, we provide this description from a low-level perspective.

We define a *write* as a memory-modifying assembly instruction, or a `call` instruction transferring control to a C standard library function that causes modification of memory outside its own stack frame or the stack frames of functions it calls.

We define an *object* as a continuous closed interval of memory addresses allocated for storing data of a single source code-defined entity or an implicitly created entity such as a saved instruction pointer. Objects can be either unitary or composite. *Unitary objects* are considered homogeneous chunks of memory for which we disregard any further structure within them. In contrast, *composite objects* are composed of further unitary or composite objects contained within their address interval. Following this definition, scalar variables of type `int` stored in memory are 4-byte unitary objects, and a pointer stored in memory is an 8-byte unitary object, just like a saved instruction pointer at the bottom of a stack frame, a stack canary value, or a spilled 64-bit register. On the other hand, we consider structures to be composite objects, as they are composed of child objects, which can either be unitary or composite themselves.

A special case are arrays. Although we might intuitively consider them composite objects in general, we argue that array-internal OOB writes are rare for arrays of primitive types, with a type confusion being the only practical way to cause them. Arrays with only unitary elements are therefore also unitary objects. In arrays of composite objects, however, array-internal OOB writes are substantially more likely to occur, as the composite objects potentially contain array fields. Therefore, we design our approach to allow us to treat arrays of composite objects either as unitary or composite objects.

Unitary objects are generally non-overlapping. On the other hand, composite objects overlap with all of their (recursive) child objects. Due to substantial difficulties with determining the active member of a union datatype at run time, we generally consider unions unitary objects, regardless of their member types. Objects are uniquely identified by their interval of encompassed memory addresses. While the lifetime of objects located in the global

program sections spans from the beginning to the end of program execution, the lifetime of objects located on the stack typically spans from the call to the function in whose stack frame they are located until the return of control flow to the calling function.

We define the *intended destination object* of a write as the object that the programmer intended to be accessed with this write, according to the source code semantics of the program. While the intended destination object can often be clearly identified for some types of writes, this is not always the case. Performing such identification is a major part of our work and is further described in Section 5.6 and Section 5.9.

The defined concepts now allow us to specify our definition of an out-of-bounds write:

A write to an interval of memory addresses is in-bounds if the write's interval is fully contained in the intended destination object's interval. Otherwise, it is out-of-bounds.

5.1.1. Scope

In general, we only strive to detect *primary* OOB writes, i.e., those that cause initial memory corruption of the program under test. It is not our priority to detect and identify the effects of *secondary* OOB writes that are only facilitated by exploiting a previous memory corruption vulnerability and therefore rely on the program already being in an invalid state with respect to the behavior intended by the programmer. We introduce this constraint for the following reasons.
Firstly, the occurrence of secondary OOB writes in most cases effectively requires the involvement of an intelligent attacker. However, as we aim to uncover capabilities from mere PoCs of OOB writes, for example discovered through fuzzing, not from weaponized exploits crafted by intelligent attackers, we are unlikely to encounter PoC inputs that cause secondary OOB writes to begin with. As an example, take a simple stack-based buffer overflow that can cause the saved instruction pointer to be overwritten. On a system with a 64-bit address space, it is highly unlikely that a PoC input discovered by a fuzzer causes the instruction pointer to be overwritten with the address of a valid instruction, jumping to which would not cause a segmentation fault or invalid instruction error. It is even more unlikely that the PoC input would cause the control flow to be diverted in a way that causes further OOB writes before crashing.
Secondly, the first OOB write in a chain of OOB writes is arguably the most decisive one, as it allows an attacker to obtain an initial foothold. The possibility for secondary OOB writes is entirely dependent on the capabilities of the first one.
Thirdly, aiming to characterize the capabilities of secondary OOB writes would, to some extent, violate our goal of keeping our capability extraction general and staying exploitation-agnostic, as it would cause us to zero in on one or more ways to utilize the discovered vulnerability for exploitation and thereby force us to move into the realm of automatic exploit generation.

We restrict the scope of program components for which we perform OOB write detection to the compiled representation of functions defined in the program's source code. This excludes any components used for dynamic linking or loading, and generally, any instructions executed before the first function of the program is called or after it returns. Furthermore, any instructions within standard library functions are out of scope. We assume that any OOB write occurring within a standard library function is due to faulty arguments passed to it during the call from the program under test and can be fully characterized by only observing those arguments. Similarly, any computation taking place within kernel space is out of scope. Furthermore, we assume that the program under test does not directly invoke system calls. This is a reasonable assumption, as C programs typically do not explicitly invoke system calls themselves but do so implicitly via standard library functions.

As indicated in Section 1.1, we restrict our scope for detecting OOB writes to those occurring in the static sections `.data` and `.bss`, as well as on the stack. Thus, we disregard OOB writes occurring within any other writable program section, such as the heap, `.got`, and `.got.plt`. However, as these sections are typically not directly interacted with by program components that were manually developed by the programmer but serve to facilitate, e.g., dynamic linking, it is reasonable to assume that a program under test does not cause OOB writes originating from these sections.

Lastly, we assume that any program under test is compiled without optimizations that either cause the frame base pointer to be omitted and the `rbp` register to be used for other purposes or cause the optimization of tail calls by jumping and re-using the existing stack frame. The motivation for these assumptions is to avoid a substantial increase in the complexity of our approach's implementation that would be necessary to handle frame pointer omission and tail call optimization. However, as these are primarily implementation issues, we argue that this assumption does not significantly decrease the generalizability of our design.

5.2. Approach Overview

We now provide a high-level description of our approach by first introducing some core concepts our approach relies on, after which we present a high-level description of our algorithm.

5.2.1. Concepts

Our design relies on a number of concepts devised by us that we will now briefly introduce and describe in detail later in this chapter.

Dependent writes The first important concept is our distinction between two types of writes contained in the machine code of the program under test. Dependent writes are characterized by the fact that the pointer to their intended destination object is computed by another instruction before the write, thereby making it necessary to detach the logic for determining the intended destination object from the logic for checking the legality of the write. This separation requires a mechanism for tainting pointers with additional information and retrieving this information from the pointer or a pointer derived from it at a later point during program execution.

Independent writes Independent writes, on the other hand, are characterized by the pointer to the intended destination object being computed within the instruction itself. As we have to assume that individual instructions appear atomic from our outside observer's view, this makes it impossible to follow the same approach as for dependent writes. However, as we will further outline in Section 5.4, independent writes always have the same intended destination object. Thus, we can nevertheless perform practical bounds checking. The categorization of writes into dependent and independent writes is explained in detail in Section 5.4.2.

Pointer-creating instructions To facilitate bounds checking of dependent writes, it is essential to taint pointers with their intended pointee object as early in their lifetime as possible. Therefore, we introduce the concept of pointer-creating instructions - machine code instructions in the program under test that cause the creation of a pointer by storing it in a register. We will further describe the exact types of instructions that we consider pointer-creating in Section 5.7.

Bounds-narrowing instructions Bounds-narrowing instructions transform a pointer to a composite object into a pointer to a (recursive) child object of that composite object. This pointer is then either dereferenced by the instruction itself, e.g., to write to a specific memory address, or saved in a register to be used by other instructions. BNIs require us to restrict the legal range written to when dereferencing the resulting pointer to the interval of the (recursive) child object. We will hereafter refer to the information on which (recursive) child object the bounds need to be narrowed to as the *bounds-narrowing target* (BNT). Further details on how to identify bounds-narrowing instructions are provided in Section 5.8.

5.2.2. Algorithm

We present a high-level description of our approach in the form of pseudocode in Algorithm 1. Operations that correspond to core challenges solved by our design are marked in orange. The given function is intended to be executed for every machine code instruction of the program under test.

As shown in the pseudocode, our approach focuses on the three types of instructions just discussed: pointer-creating instructions (PCIs), bounds-narrowing instructions (BNIs), and writes.

If the instruction is identified as a write, which, despite being subject to some restrictions in our design as outlined in Section 5.3, can be considered an implementation challenge, the destination address and the number of bytes written are determined. Now, depending on whether the write is dependent or independent - the distinction between which constitutes the first challenge addressed by our design - we determine the intended destination object of the write in one of two ways.
If the write is dependent, we obtain the taint from the destination address, which we previously added when handling the corresponding pointer-creating instruction and potentially modified by one or more bounds-narrowing instructions. Then, the intended destination object can simply be determined from the taint. However, the dependent write might at the same time act as a bounds-narrowing instruction. Recognizing an instruction as a bounds-narrowing instruction and identifying the target to which bounds need to be narrowed is the second challenge we address in our design. If the dependent write is also a bounds-narrowing instruction, we obtain the corresponding bounds-narrowing target and change the intended destination object accordingly.
If the write is independent, it does not use a destination pointer that has previously been created by a pointer-creating instruction and can therefore not be bounds checked in the same fashion as dependent writes. However, as the intended destination object of independent writes is guaranteed to be static over the entire runtime of the program under test, it can be determined through static analysis - the third challenge addressed by our design.
After having identified the intended destination object in either case, we can easily verify whether the write is in-bounds from the intended destination object, the destination address, as well as the number of bytes written.

The recognition of an instruction as a pointer-creating instruction is the fourth challenge we address in our design. If an instruction that is not a write is identified as a pointer-creating instruction or a bounds-narrowing instruction (or

both), we first obtain the bounds-narrowing target of the instruction. Naturally, this will be trivial for an instruction that is not bounds-narrowing. Afterwards, we obtain the resulting pointer as it is stored in a register by the instruction. Using this pointer, as well as the bounds-narrowing target, we identify the intended pointee object of this pointer, which constitutes the fifth challenge. This is facilitated by our knowledge of the program's memory layout at run time, the acquisition of which corresponds to the sixth and last challenge.
Finally, we taint the pointer with information about the intended pointee object using a pointer tainting mechanism, which is primarily an implementation challenge and therefore not further described in our design.

Algorithm 1 High-level pseudocode description of our approach

```
1:  procedure ProcessInstruction(inst)
2:      if inst is write then
3:          dstAddr ←getWriteDstAddr(inst)
4:          nBytes ←getWriteBytesNum(inst)
5:          if inst is dependent write then                          ▷ see Section 5.4
6:              taint ←getPointerTaint(dstAddr)
7:              ido ←getObjectFromTaint(taint)
8:              if inst is bounds-narrowing instruction then         ▷ see Section 5.8
9:                  bniTarget ←getBoundsNarrowingTarget(inst)        ▷ see Section 5.8
10:                 ido ←narrowObject(ido, bniTarget)
11:             end if
12:         else                                                     ▷ independent write
13:             ido ←getObjectFromIndependentWrite(inst)             ▷ see Section 5.6
14:         end if
15:         if [dstAddr, nBytes - 1] is not a subrange of [ido.start, ido.end] then
16:             reportOOBw()
17:         end if
18:     else if inst is pointer-creating or bounds-narrowing instruction then   ▷ see Section 5.7 and Section 5.8
19:         bniTarget ←getBoundsNarrowingTarget(inst)                ▷ see Section 5.8
20:         ptr ←getResultingPointer(inst)
21:         ido ←getPointeeObject(ptr, bniTarget)                    ▷ see Section 5.9
22:         taintPointer(ptr, ido)
23:     end if
24: end procedure
```

To summarize, our design is subject to the following six challenges. We address all of them in the upcoming sections of this chapter.

1. Distinguishing dependent from independent writes (Section 5.4)
2. Identifying bounds-narrowing instructions and their targets (Section 5.8)
3. Identifying intended destination objects of independent writes (Section 5.6)
4. Identifying pointer-creating instructions (Section 5.7)
5. Extracting memory layout at run time (Section 5.5)
6. Determining intended pointee objects from pointers and bounds-narrowing targets (Section 5.9)

5.3. Identifying Writes

A vital prerequisite for our further approach is to define what we consider a write, both at the assembly and the intermediate representation level. Although intuitively, every instruction that in some way, explicitly or implicitly, modifies memory must be considered a write, we argue that this introduces unnecessary complexity and considering only a subset of these instructions is sufficient for performing OOB write detection.

For brevity, we will hereafter refer to writes contained in the compiled binary in the form of AMD64 instructions as *ASM writes*, while we will refer to writes contained in the IR in the form of LLVM IR instructions as *IR writes*.

5.3.1. ASM Level

With AMD64 being a CISC ISA, there exists a multitude of different instructions that explicitly or implicitly modify memory. In practice, however, only a small subset of instructions is responsible for most writes to memory, of which only some are capable of causing an OOB write. Our observations show that almost all memory-modifying instructions in programs compiled from C source code using Clang are either part of the `mov` family[1], a `push` instruction or a `call` instruction. We can safely disregard the latter two, as they will generally only write to the top of the stack, i.e., relative to `rsp`. Thus, a manipulated stack pointer is a prerequisite for a `push` or `call` causing an OOB write. As it is safe to assume that a manipulated stack pointer only occurs after a preceding memory corruption, an OOB write by a `push` or `call` instruction is out of scope for us, as outlined in 5.1.1.

Therefore, we define non-innocuous writes, hereafter simply referred to as writes, as instructions that

a) are part of the `mov` family of instructions, and

b) have a memory location as their destination operand.

We refer to instructions that modify memory but cannot cause an OOB write according to our scope as *innocuous writes*.

5.3.2. IR Level

Compared to the AMD64 instruction set, the LLVM IR consists of a relatively small set of memory-modifying instructions. The LLVM language reference [34] specifies around ten such instructions, although further instructions can potentially be translated to machine code that modifies memory. An example of this are `call` instructions, which in LLVM are not bound to modify memory, but will be translated to `call` and `push` instructions in assembly. However, we found that almost all AMD64 instructions that we consider writes, as outlined above, are translated from one of only three LLVM IR instructions: `store`, `llvm.memcpy`, and `llvm.memset`. The latter two are not pure instructions but intrinsic functions, which are functionally similar to their well-known counterparts in the C standard library. However, they do not have a return value and are therefore frequently translated directly to AMD64 instructions by the backend in order to avoid costly function calls, making them potential candidates for becoming ASM writes.

5.4. Distinguishing Writes

We define categorizations of writes at two different levels. On the IR level, we categorize writes into *static writes* and *dynamic writes*. On the assembly level, we categorize writes into *dependent writes* and *independent writes*, the latter of which is again split into innocuous and non-innocuous independent writes.
The motivation for categorizing into dependent and independent writes is to provide a clear division between two different ways of computing destination addresses of memory-modifying instructions requiring entirely disparate approaches for bounds checking.
On the other hand, the motivation for categorizing into static and dynamic writes is to provide a higher-level division between writes that, although not corresponding exactly to the low-level division, eases determining intended destination objects by creating the relationships between writes on the IR and ASM level presented in Figure 5.1.

5.4.1. IR Level: Static Writes vs. Dynamic Writes

On the IR level, we define *static writes* as those write instructions for which the intended destination object is fixed at compile time, while *dynamic writes* are those write instructions for which the intended destination object is determined at run time and therefore typically depends on the control flow of the program. It is important to note that, for determining whether a write is static or dynamic, we operate on a function-level scope. Therefore, even if the program's control flow causes a write to have the same intended destination object on every possible control flow path through the program, we still consider it a dynamic write if this behavior cannot be identified from the function that contains the write. An example of this is presented in Figure 5.2. Although the write in line 20 will only ever write to `buf1` allocated in the stack frame of `main`, we consider it a dynamic write, as this cannot be concluded by inspecting only the containing function `func`.

[1] This effectively includes all instructions containing the `mov` string in their mnemonic, such as `cmovnc`, `movaps`, or `kmovq`.

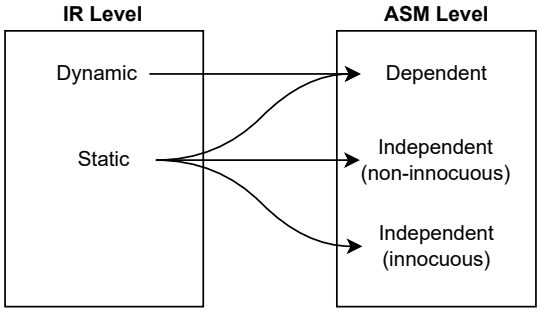


Figure 5.1: Relationships between write categories on ASM and IR level

Further examples of static and dynamic writes are presented in Figure 5.2. For better readability, we give these examples in the form of C code that, if compiled directly to LLVM IR without optimizations, would result in the annotated type of write. For most writes in the example, the rule of thumb applies that a write is static if it is performed to an object located in a global section or within the stack frame of the containing function. An exception to this is the write in line 23. Here, the preceding condition makes it impossible to know at compile time whether the write will be performed to `buf3`, allocated in the same stack frame, or to `buf1`, allocated in the previous stack frame. Thus, the write is dynamic.
Note that the given code is merely for demonstration purposes and does not perform any specific task. In fact, the code can be considered unsafe, as it uses the values of variables prior to their initialization.

We provide details on our procedure for analyzing LLVM IR to determine whether a write is static or dynamic in Section 5.6.

```
1  char globbuf[64] = { 'Y' };
2
3  int main(int argc, char** argv) {
4    char buf1[64];
5    char buf2[64];
6
7    buf1[42] = 'A';                  // static write
8    if (argc > 1)
9      buf1[argc] = **argv;           // static write
10   func(buf1, buf2);
11 }
12
13 void func(char * buf, char* idxbuf) {
14   char buf3[32];
15   char* bufPtr;
16
17   bufPtr = buf3;                   // static write
18   bufPtr[0] = 'A';                 // static write
19
20   buf[0] = 'A';                    // dynamic write
21   if (*buf == 'A')
22     bufPtr = buf;                  // static write
23   bufPtr[0] = 'A';                 // dynamic write
24
25   if (*idxbuf < 128)
26     globbuf[*idxbuf] = 'A';        // static write
27   else
28     buf3[*idxbuf] = 'A';           // static write
29 }
```

Figure 5.2: Examples of writes in C that would become static and dynamic writes when compiled to LLVM IR

5.4.2. ASM Level: Dependent Writes vs. Independent Writes

On the machine code level, *dependent writes* modify memory at a location specified by a pointer that was previously created by a pointer-creating instruction. The crucial characteristic of dependent writes is that their intended destination object is determined not within the instruction itself but by a preceding pointer-creating instruction. This forces us to detach the identification of the intended destination object from the bounds check of the actual write and thereby requires a mechanism for associating additional data with pointers to be in place, akin to the techniques for augmenting pointers discussed in Section 3.2.

On the other hand, *independent writes* modify memory at a location that is only calculated within the instruction itself, independent of any pointer-creating instructions. Components from which the destination address is computed include stack frame boundary pointers (`rsp` and `rbp`), the instruction pointer, immediate addresses, immediate offsets, as well as offsets contained in registers. Thus, an important characteristic of independent writes is that their intended destination objects can usually not be reliably inferred from the results of previous instructions in combination with knowledge about the locations of objects in memory. Instead, the intended destination objects are only determined within the instructions themselves, which constitutes the fundamental difference to dependent writes and makes the separate treatment of independent writes necessary. The insight that enables us to determine the intended destination objects of independent writes regardless is that any independent write is also a static write. Independent writes occur in some cases when writing to an object located in the global sections, e.g., a static variable, and in most cases when writing to an object located within the current function's stack frame, e.g., an automatic variable.

We further divide independent writes into two subcategories - innocuous independent writes and non-innocuous independent writes. The former is - just like innocuous writes such as `push` and `call` instructions - not capable of causing an OOB write in our scope. Non-innocuous writes, on the other hand, can potentially cause OOB writes and must therefore be checked by our approach.

We define innocuous independent writes as those instructions of the `mov` family whose destination operand is a memory address calculated in a way specified by one of the rows in Table 5.1. Clearly, no such instruction can cause an OOB write in our scope - a write with an immediate destination address cannot do so under any condition - just like a RIP-relative write. Likewise, a write with a destination address that is only computed from a stack frame boundary register (`rbp` and `rsp`) can only cause an OOB write when a preceding memory safety violation modifies the corresponding register.

Addressing mode	Restriction	Example
Disp	none	`mov [0x403020], rcx`
Base	Base ∈ {rbp,rsp}	`mov [rsp], rcx`
Base + Disp	Base ∈ {rbp,rsp}	`mov [rbp - 0x30], rcx`
RIP + Disp	none	`mov [rip + 0x1430], rcx`

Table 5.1: Destination operands of writes that we consider to be innocuous independent writes

We define non-innocuous independent writes as those instructions of the `mov` family whose destination operand is a memory address calculated in a way specified by one of the rows in Table 5.2.
Any write for which the base or index is given by a stack frame boundary register can be expected to write to a fixed object within the current function's stack frame. Likewise, writes in which the displacement is a pointer to a static memory section (`.data` or `.bss`) can be expected to write to a fixed object within the static memory sections. However, as any destination operand satisfying the given conditions uses at least one additional variable component to calculate the actual address, this component can potentially cause the write to become out-of-bounds. Thus, any such write is non-innocuous.

Addressing mode	Restriction	Example
Base + Index	Base or Index ∈ {rbp,rsp}	`mov [rsp + rax], rcx`
Base + Disp	Disp points to static segment	`mov [rax + 0x403020], rcx`
Base + Index + Disp	Base or Index ∈ {rbp,rsp}	`mov [rsp + rax + 0x20], rcx`
Base + Index + Disp	Disp points to static segment	`mov [rax + rbx + 0x403020], rcx`
Base + (Index * Scale)	Base ∈ {rbp,rsp}	`mov [rsp + (rax * 4)], rcx`
(Index * Scale) + Disp	Disp points to static segment	`mov [(rax * 2) + 0x403020], rcx`
Base + (Index * Scale) + Disp	Base ∈ {rbp,rsp}	`mov [rsp + (4 * rax) + 0x10], rcx`
Base + (Index * Scale) + Disp	Disp points to static segment	`mov [rbx + (4 * rax) + 0x403020], rcx`

Table 5.2: Destination operands of writes that we consider to be non-innocuous independent writes

5.5. Memory Layout Extraction

Maintaining information on which objects occupy which memory regions is an essential prerequisite for performing non-invasive OOB write detection and mapping regions affected by an OOB write to their corresponding objects. Debug data in the DWARF format, attached to the binary file during compilation, can be used to obtain a large amount of information necessary for this.

Any memory object described in a DWARF debug information entry (DIE) represents either a program variable, a formal parameter such as a function argument, or a constant. Each DIE contains a number of attributes describing properties like the source code-level name of the object, its type, as well as its location at run time. Further information can be inferred from the ancestors of the DIE in the tree-like hierarchical arrangement of DIEs. This includes information about the object's scope and, in the case of an automatic variable, the stack frame base register.
As the location descriptions we extract from the object DIEs contain solely the location at which the first byte of the object is located but no further information on its size or the internal structure of composite objects, we are forced to extract such information from the type DIEs referenced by the object DIEs.

In the debug information, the locations of objects in the global sections `.data` and `.bss` are generally described by absolute memory addresses, which, together with the fact that these objects are present over the entire execution time of the program, makes their extraction and maintenance simple. For objects located on the stack, however, the location is generally specified relative to a stack frame boundary register (`rbp` or `rsp`). This not only forces us to obtain this register value at run time to compute the absolute location of the object but also makes it necessary to permanently keep track of the call stack at run time.

Call stack tracking at run time can be performed by monitoring for `call` and `ret` instructions and combining the address to which the control is transferred by the `call` instruction with information on the entry location of each function, contained in the corresponding DWARF DIEs. As indicated in Section 5.1, we assume that no tail call optimizations are employed in the compiled program, and monitoring for `call` and `ret` instructions is therefore sufficient to recognize the transfer of control flow to another function. However, the modification of our approach to apply to programs in which jumps are performed instead of `call` and `ret` instructions is certainly possible, with the main drawback being an increased implementation complexity.

5.5.1. Effects of Optimizations

The placement of objects in memory, and therefore the information obtained from an analysis of the corresponding DIEs, is heavily influenced by the level of optimizations applied during compilation. In many regards, an entirely unoptimized program provides us with the best conditions for accurately extracting its memory layout and performing OOB write detection. This is because the placement of data in memory follows a relatively simple and predictable pattern in this case, with almost all variables, formal parameters, and constants being located in memory at a single, well-defined location that does not change at run time. Compile-time optimizations, however, cause memory layout extraction to become substantially more challenging by influencing the object placement in memory in the following ways.

Moving objects to registers: Motivated by the substantially faster access to registers than memory, some objects do not have a representation in memory and are maintained solely in registers.

Removing objects entirely: Optimizations might entirely remove objects from the program if they are unused or can be represented implicitly.

Moving objects over time: The low number of available general-purpose registers can cause an object not to be stored in a register over its entire lifetime but to be temporarily spilled to memory. Once the object is loaded back into a register, a different register than before might be used, causing it to occupy different locations in registers and memory over its lifetime.

Fragmenting objects: Optimizations occasionally cause composite objects to be split into multiple fragments, which then reside in different locations in registers, memory, or are omitted entirely. A structure consisting of two integers and an array, for example, might be split into three parts, with the first integer stored in a register, the array stored in memory, and the second integer omitted entirely after being deemed superfluous during optimizations.

However, with each DIE of a program variable, formal parameter, or constant containing detailed information about the object's location at run time, we can handle most effects of optimizations reasonably well.
If the location attribute of an object indicates that, over its entire lifetime, it exists only in registers or is not explicitly represented at all, we can simply ignore this object as it can never be the source of an OOB write, and neither can it be modified by one.
If the object is not located in the same memory location over its entire lifetime, the location information contained in the DIE provides, for each location of the variable, the address of the first and last instruction in the program at which the variable is located in a specific register or memory location. This makes it necessary for us to record the

positions of variables in a spatial dimension and a temporal dimension.
Fragmentation, however, cannot be handled as cleanly as the previously discussed effects of optimization. The reason for this is that, although a fragmented variable's DIE describes the positions of fragments in memory and their size, it does not provide information on which fragment contains which part of the composite object. As such, we are unable to perform bounds-narrowing on fragmented composite objects.

An example of a heavily fragmented variable's DIE, with fragments being located in different locations at different points during the variable's lifetime, is presented in Figure 5.3. The variable, a structure consisting of three 1-byte fields, is fragmented into these three parts, one of which is located in the register `rsi` at the first location description. At the following location description, one fragment, of which it is not clear whether it is the same as before, is located on the stack at `rbp-56`. In the fourth location description, all three fragments have a location, one of them on the stack and the remaining two in registers.

```
DW_TAG_variable
  DW_AT_name  ("back")
  DW_AT_decl_file ("libpng/libpng/pngrtran.c")
  DW_AT_decl_line (1608)
  DW_AT_type  (0x00016403 "png_color")
  DW_AT_location  (0x0002242a:
     [0x415ee9, 0x415eec): DW_OP_reg4 RSI, DW_OP_piece 0x1
     [0x415eec, 0x415ef8): DW_OP_breg6 RBP-56, DW_OP_piece 0x1
     [0x415ef8, 0x415f04): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1
     [0x415f04, 0x415f1d): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1, DW_OP_reg9 R9, DW_OP_piece 0x1
     [0x4160cc, 0x4160d0): DW_OP_reg12 R12, DW_OP_piece 0x1
     [0x4160d0, 0x4160d7): DW_OP_breg6 RBP-56, DW_OP_piece 0x1
     [0x4160d7, 0x4160de): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1
     [0x4160de, 0x41610f): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1, DW_OP_reg9 R9, DW_OP_piece 0x1
     [0x4168a0, 0x4168f7): DW_OP_reg12 R12, DW_OP_piece 0x1
     [0x4168f7, 0x416936): DW_OP_reg12 R12, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1
     [0x416936, 0x41693b): DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1
     [0x41695f, 0x4169b4): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1, DW_OP_reg9 R9, DW_OP_piece 0x1
     [0x4169b4, 0x416a7b): DW_OP_breg6 RBP-56, DW_OP_piece 0x1
     [0x416a7b, 0x416a9e): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_reg11 R11, DW_OP_piece 0x1, DW_OP_reg9 R9, DW_OP_piece 0x1
     [0x416a9e, 0x416bf3): DW_OP_breg6 RBP-56, DW_OP_piece 0x1, DW_OP_piece 0x1, DW_OP_reg9 R9, DW_OP_piece 0x1)
```

Figure 5.3: Example for the DIE of a fragmented composite object whose fragments change location over time

5.6. Identifying Intended Destination Objects of Independent Writes

Independent write checking is based on the insight that, as any independent write corresponds to a static write in the IR, its intended destination object is always the same, regardless of the program's control flow. However, due to the loss of type and variable information incurred during compilation, reliably determining intended destination objects at independent write locations only from the information available in the binary is difficult, even if full debugging information is supplied.

Therefore, we resort to an analysis of the LLVM IR as it exists after potential optimization passes, immediately before the transformation to assembly. Although one might argue that, by requiring access to the IR, we restrict the usability of our approach, it is important to note that, for providing easily interpretable output containing the variables affected by an OOB write, we inherently require access full debug information already. As off-the-shelf binaries containing full debug information but without available source code are exceptionally rare, we argue that, if full debug information is available, the source code is in most cases also available, and therefore also the IR.
Although alternative approaches for determining intended destination objects of independent writes are conceivable, some of which would only require access to the compiled binary with debug information, none of these approaches are practical for us. The reasons for this primarily lie in the high complexity and error-proneness of such alternative approaches.

By leveraging information from the LLVM IR, we avoid having to perform direct matching between independent writes and memory objects and instead take a detour by matching to static writes and IR destination variables, as displayed by the solid arrows in Figure 5.4. This poses the following three challenges, each of which corresponds to one step in Figure 5.4. We describe our approach to solving each of the three challenges in the remainder of this section.

1. Matching each independent write in ASM to its corresponding static write in the IR
2. Determining the destination variable for each static write in the IR
3. Matching each identified destination variable of the IR to its corresponding object in memory

As described in Section 5.3, innocuous independent writes can never cause an OOB write in our scope. Therefore, we disregard innocuous independent writes entirely and only attempt to match non-innocuous independent writes to their intended destination objects.

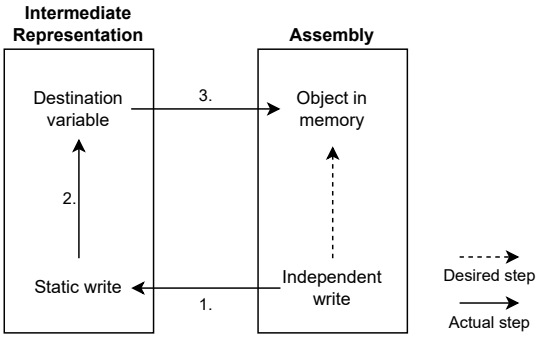


Figure 5.4: Approach for determining the intended destination object of an independent write

Matching Independent Writes to Static Writes

The step of matching independent writes to static writes inherently relies on our ability to identify both independent writes in assembly, and static writes in the IR, as described in Section 5.3.

In practice, the mapping from ASM writes to IR writes is rarely ever bijective. This is not only due to our disregard for certain memory-modifying instructions in ASM and the IR, but also caused by the inherent differences between AMD64 assembly and the LLVM IR, which occasionally causes writes in the IR to be split up into multiple writes in ASM. As such, we aim to match independent writes to static writes on a best effort basis, knowing that we will, in some cases, be unable to find a match for an independent write and, in rare cases, match it to an incorrect static write.
However, our categorization of writes outlined in Section 5.4 allows us to substantially reduce the search space when matching an independent write to its corresponding static write in the IR by excluding any dynamic write as a potential match.

At its core, our approach for matching independent to static writes relies on line number debug information. This information is present both in the IR in the form of metadata attached to instructions and in the ELF file in the form of DWARF debug information, as outlined in Section 2.6. A line number debug entry for an instruction at a specific address comprises the source file name, line number, as well as column number at which the source code corresponding to the instruction is located. While the location information conveyed by such line number debug entries is entirely irrelevant to our approach, we can leverage these distinctive features of instructions to find the static write representation of an independent write in the IR by selecting the static write with identical line number debug information.
However, not every instruction in the compiled program has a unique source code location associated with it. This is not only due to the differences in expressiveness between C, intermediate representations, and assembly, but also due to hardware limitations, macro expansions, as well as optimizations. Particularly the latter substantially influences the quality and quantity of line number debug information in the binary, with more optimizations commonly leading to more duplicate line number debug entries. We will outline our approach to alleviate this issue in Chapter 6.

Determining IR Write Destinations

After identifying the IR write corresponding to each relevant ASM write, we seek to identify the IR variable written to by this IR write. While the analog of this step would be rather challenging to perform on an ASM level due to the missing notion of variables, the static single assignment form of the IR makes this a straightforward endeavor.
We will hereafter use the term *declaration* to refer to an instruction, operator, or global variable declaration that results in the creation of an IR variable.

Starting from the operand of the memory-modifying IR instruction that specifies the write destination, we can identify the source of this value - and thereby the destination of the write - by following its declaration chain upwards, as presented in Figure 5.5. Within such a declaration chain, we can encounter four different categories of declarations, which we will now describe further.

Simple traversable declarations take a variable as an operand and result in another variable derived from the operand. Both variables contain pointer information and, in most cases, are pointer types. The intuition behind simple traversable declarations is that the input operand always points to the same object as the resulting variable, thus allowing us to continue tracing the declaration chain upwards. Traversable declarations generally do not access memory. The set of traversable declarations we identified consists of all casting instructions and operators. Although the usage of other traversable declarations, such as arithmetic instructions, in declaration chains is conceivable, we did not observe them being generated by Clang in practice.

Bounds-narrowing traversable declarations are different from their simple counterparts in the way that the input operand does not point to the same object as the resulting variable but to a composite object that constitutes

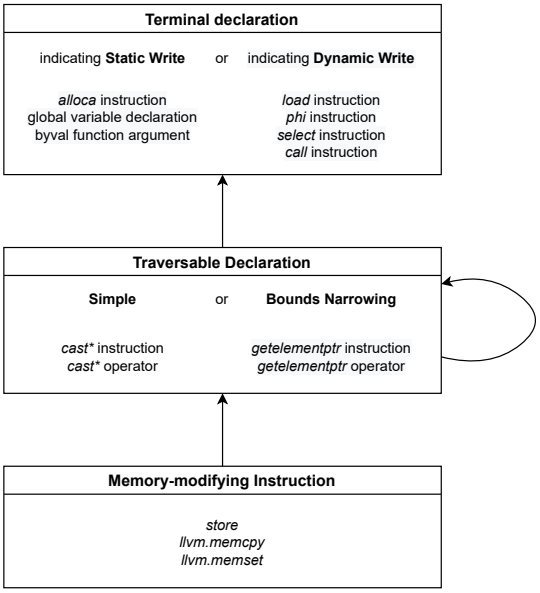


Figure 5.5: Structure of declaration chains in the LLVM IR

a (recursive) parent of the input operand object. Thus, when traversing a bounds-narrowing instruction during the search for the intended destination object of a write, it is essential to keep track of the child of the composite object that is the pointee after the instruction. In the LLVM IR, bounds-narrowing is generally performed by the `getelementptr` instruction and operator.

Terminal declarations indicating a static write are those declarations at which tracing cannot be continued, as the declaration is responsible for the definition of the object that is written to. The set of such terminal declarations includes `alloca` instructions, global variable declarations, as well as declarations of non-primitive function arguments passed by value. Invocations of the `alloca` instruction serve to declare space for an automatic variable in the current stack frame and return a pointer to it. They constitute the only way to allocate space for automatic variables. Global variable declarations, indicated by the prefix @ in the textual IR, serve to declare any static data item of the program, which, if the IR is built into an ELF file, would be placed into one of its static sections. As such, global variable declarations are also used to declare constants with global or local scope and static variables with local scope.
If a declaration chain ends in such a declaration, we can be certain that the memory-modifying instruction at the start of the chain only writes to the object corresponding to the terminating declaration. As such, it satisfies our definition of a static write.

Terminal declarations indicating a dynamic write are those declarations at which tracing cannot be reasonably continued, i.e., there does not exist a single declaration within the function scope constituting the subsequent declaration in the chain, but for which there is also no unambiguously identifiable object to which the pointer resulting from the declaration points.
The set of terminal declarations indicating a dynamic write includes the `load`, `phi`, `select`, and `call` instructions, as well as declarations of primitive function arguments passed by value and non-primitive function arguments passed by reference.
If a chain ends in such a declaration, we cannot determine the object written to by the memory-modifying instruction at the start of the chain from information available at compile time. As such, IR writes whose declaration chain ends in one of these terminating declarations satisfy our definition of dynamic writes.

An example of a declaration chain of a static write is presented in Figure 5.6. Starting from the static write in line 9, which writes to the pointer specified by the variable `%80`, the chain continues to the traversable `bitcast` instruction in line 8, which serves to transform the 8-bit pointer `%overflow_ptr285` to a 64-bit pointer, as it is taken by `store`. The `%overflow_ptr285` pointer, in turn, is created by the `getelementptr` instruction in line 7, which in this case returns a pointer to the third field of the `%struct.char_payload` structure pointed to by the `%payload` pointer. Finally, inspecting the declaration of the `%payload` pointer reveals that it is the result of the `alloca` instruction in line 5, which returns a pointer to a `%struct.char_payload` structure allocated on the stack. Thus, it can be concluded that the write in line 9 is static.

```
%struct.__jmp_buf_tag = type { [8 x i64], i32, %struct.__sigset_t }
%struct.__sigset_t = type { [16 x i64] }
%struct.char_payload = type { i32, i64, i8*, i8*, [1 x %struct.__jmp_buf_tag]*, i8*, i8*, i64, i64, i8* }
...
%payload = alloca %struct.char_payload, align 8
...
%overflow_ptr285 = getelementptr inbounds %struct.char_payload, %struct.char_payload* %payload, i64 0, i32 2
%80 = bitcast i8** %overflow_ptr285 to i64**
store i64* %arrayidx284, i64** %80, align 8
```

Figure 5.6: Example of a declaration chain ending in a memory-modifying IR instruction

Matching IR Variables to Memory Objects

The third step of determining the intended destination object for static ASM writes - matching the IR variables to corresponding memory objects - is arguably the simplest one. Having found the static variable declarations and automatic variable creations via `alloca` in the previous step, LLVM allows for easily obtaining the name of the corresponding variable in the source code. Although static writes can also write to objects not defined in the source code, e.g., IR variables created to store spilled register contents, we found that such writes are typically transformed into innocuous independent ones. Thus, any IR variable we seek to match is source code-defined and has a name. The presence of source code-defined variable names in the DWARF debug information, along with the corresponding object's absolute or relative location in memory, allows us to perform simple name-based matching to find the corresponding variable in the debug information for each non-innocuous independent write in the compiled code. If bounds-narrowing instructions were encountered while traversing the declaration chain in the previous step and the intended destination object is therefore not the entire variable but only a child of it, we can leverage the information obtained while handling the bounds-narrowing instruction to determine this child. However, this is only possible if the variable is not fragmented. If it is fragmented, we cannot reliably determine the child's position in memory, as outlined in Section 5.5.1.

5.7. Finding Pointer-Creating Instructions

A crucial characteristic of dependent writes is the separation of the location at which the intended destination object is computed from the location at which the actual write is performed. We bridge this gap by tainting the created pointer to facilitate obtaining information about the intended destination object upon dereferencing it. As we cannot effectively leverage information from the LLVM IR to determine the intended destination object of a pointer upon its creation, we rely on the pointer's value and information about the memory layout for this.

Naturally, the likelihood of the pointer being modified to point outside of the intended pointee object, thus facilitating an OOB write, increases with the number of instructions executed after the creation of the pointer. To minimize the risk of this occurring prior to tainting, tainting is best performed immediately after pointer creation. We identified two AMD64 instructions that we found to be responsible for the greater part of pointer creations in machine code generated by the Clang compiler.

The first such instruction is *Load Effective Address* (`lea`). This instruction computes the effective address of a memory location described by the second operand and places it in the general-purpose register specified by the first operand. Although the second operand is specified as a memory location, `lea` does not access memory but only performs arithmetic [54]. While some uses of `lea` could be replaced by other individual instructions, `lea` often facilitates performing operations that would normally take multiple instructions within a single instruction, such as the operations in lines 2-4 of Figure 5.7. This is because `lea` allows the source operand to use any valid ModRM or SIB addressing mode, corresponding to all nine addressing modes presented in Section 2.1. As such, it allows for performing addition with up to three operands and does not restrict the destination operand register to one of the source registers. Furthermore, it does not modify the flags register.
These advantages cause the `lea` instruction to be used for the greater part of cases where a pointer is created from a fixed or variable offset to an address-containing register such as `rbp` or `rsp`, as it takes place when a pointer to an object on the stack is created. Note that a `lea` instruction can only be pointer-creating if its result is placed in a 32-bit or 64-bit general-purpose register. As the lower 2^{22} bytes of the virtual address space of a program running in 64-bit mode typically remain unused, storing any valid memory address will require at least a 32-bit general-purpose register.

```
lea rax, [rcx]
lea rax, [rcx+rdi]
lea rax, [rcx + 8 * rdi]
lea rax, [rcx + 4 * rdi + 23]
mov edi, 0x409678
```

Figure 5.7: Examples of pointer-creating instructions

The second type of AMD64 instruction we consider to be pointer-creating are instructions of the `mov` family with an immediate source operand that represents a memory address contained within any static program section of the program under test. An example of this is given in line 5 of Figure 5.7. The compiler frequently generates such instructions when pointers to objects whose location is known at compile time are created, as is the case for objects located in the static memory sections `.data` and `.bss`.

Although these two types of instructions are responsible for the greater part of pointer creations in AMD64 code generated by the Clang compiler, there are cases in which we will miss a pointer creation when only monitoring these instructions. Examples of this are cases where the pointer's address is already present in a register before the pointer is created. For example, this occurs when a pointer to an object located at the very top of the stack frame is created, with its address equal to `rsp`.
The consequences of such missed pointer creations can, to some extent, be contained by employing additional checks at locations where pointers are likely to leave the function scope. At such locations, we can, for each potential pointer that is not tainted, check whether it points to an existing memory section and, if so, taint it. Locations at which this can be performed include function argument registers during function calls, the return value register upon returning from a call, and any writes to memory whose size matches the size of a pointer. However, such delayed tainting is arguably more prone to incorrect identification of the intended pointee object, as the larger distance between the pointer creation site and the tainting site gives more room for modifications of the pointer. On the other hand, mistakenly tainting non-pointer data is largely unproblematic, as the taint of a piece of data is only considered when it is dereferenced as a pointer.
A more substantial issue than missed pointer tainting is incorrect pointer tainting. When a pointer is already out-of-bounds upon its creation, our approach will potentially identify an incorrect intended destination object with which the pointer is tainted. A dependent write relative to this incorrectly tainted pointer will then, with high certainty, either cause the approach to miss one or more OOB writes or cause false positives. Either can only be uncovered through tedious manual analysis.
An example in which a pointer can already be out-of-bounds upon creation is the case in which a pointer to a variable element in a stack-allocated integer buffer is created, e.g., via `&buffer[x]`, with `x` being variable and potentially larger than the size of the buffer. With enabled optimizations, this would for example be translated to `lea rdi, [rbp + 4 * rax - 64]`, with `rax` containing the value of `x`.

5.8. Finding Bounds-Narrowing Instructions

Performing accurate bounds-narrowing is an inherent requirement for correctly detecting intra-object OOB writes with a pointer-based approach like ours. This is because writes to child objects of composite objects, such as structure fields, are often performed by dereferencing a modified pointer created initially to point to the base of the composite object. For our previously described approach for tracking pointers, this would mean that a pointer to such a child object would inherit the taint - and therefore the bounds information - of the pointer to its parent object. Thus, a write dereferencing the pointer to the child object would not be detected as OOB if it modifies memory at a location outside of the child object's region, as long as the modifications stay within the region of the parent object.
To exemplify this, consider the example in Figure 5.8. Upon being passed to `createUser`, the pointer `user` will have a taint that allows its usage for writing into the memory region of its pointee object of type `userEntry`. Without bounds-narrowing, this taint will be propagated to the pointer passed to `fgets` in line 13, allowing the call to modify the entire memory region of the `userEntry` structure without detection.
An important assumption in our handling of pointer bounds is that a pointer's bounds can be narrowed by a BNI but can never be widened again. Thus, at each instruction in the program, a pointer's bounds can either remain the same or be narrowed to a sub-interval of its old bounds.

From compiled code, locations at which bounds-narrowing needs to be performed are usually difficult to identify. Consider the compiled version of the `createUser` function in Figure 5.9. In the first call to `fgets` in line 18, the pointer `user`, located at `rbp-8`, is directly loaded into `rdi` at line 15 to be passed as the first argument, although not the pointer to the entire structure but just to the first field is to be passed. As the `username` field is located at the very beginning of the structure and therefore starts at the same address as the entire structure, this behavior is to be expected. However, it leaves no trace in the compiled code that indicates the need to narrow its bounds.
In the second call to `fgets` on line 25, on the other hand, bounds-narrowing is evident by the addition of a fixed offset to the pointer in line 22.
The absence of any clear indicator for a need to perform bounds-narrowing, as is the case for the first `fgets` call, is already a significant obstacle for identifying bounds-narrowing sites in compiled code. However, we observed several other patterns in compiled code that represent bounds-narrowing, particularly in highly optimized code, making this even more of a challenging endeavor. Thus, we again resort to analyzing the LLVM IR to make bounds-narrowing feasible.

```c
struct userEntry {
  char username[32];
  char password[32];
  int id;
  bool isAdmin;
};
typedef struct userEntry userEntry;

void createUser(userEntry *user) {
  user->id = getNewUserId();
  user->isAdmin = false;
  puts("Enter username: ");
  fgets(user->username, sizeof(userEntry), 0);
  puts("Enter password: ");
  fgets(user->password, sizeof(userEntry), 0);
}
```

Figure 5.8: Example of intra-object OOB write that can only be detected when performing bounds-narrowing

```asm
createUser:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     qword ptr [rbp - 8], rdi
    mov     al, 0
    call    getNewUserId
    mov     ecx, eax
    mov     rax, qword ptr [rbp - 8]
    mov     dword ptr [rax + 64], ecx
    mov     rax, qword ptr [rbp - 8]
    mov     byte ptr [rax + 68], 0
    movabs  rdi, offset .L.str.3
    call    puts
    mov     rdi, qword ptr [rbp - 8]
    mov     rdx, qword ptr [stdin]
    mov     esi, 72
    call    fgets
    movabs  rdi, offset .L.str.4
    call    puts
    mov     rdi, qword ptr [rbp - 8]
    add     rdi, 32
    mov     rdx, qword ptr [stdin]
    mov     esi, 72
    call    fgets
    add     rsp, 16
    pop     rbp
    ret
```

Figure 5.9: Compiled `createUser` function without optimizations

5.8.1. IR Level

In the LLVM IR, any pointer manipulation is performed via the `getelementptr` (GEP) instruction or its corresponding operand[2]. As such, we can identify all bounds-narrowing sites in the IR by inspecting GEP instructions for whether they convert a pointer to a composite object (as defined in Section 5.1) into a pointer to a child of that composite object.

An example of such a GEP instruction constituting a bounds-narrowing site, generated from the C code in Figure 5.10, is presented in Figure 5.11. The GEP instruction converts a pointer to a structure of type `B` into a pointer to a field in a structure contained in an array within it.

```c
struct A {
  char a1[32];
  int a2;
};

struct B {
  int b1[8];
  struct A b2[4];
};

int fun(struct B *sB) {
    return sB->b2[2].a2;
}
```

Figure 5.10: Example of bounds-narrowing in nested composite object

```
%struct.A = type { [32 x i8], i32 }
%struct.B = type { [8 x i32], [4 x %struct.A] }

define dso_local i32 @fun(%struct.B* %0) #0 {
  %2 = getelementptr %struct.B, %struct.B* %0, i64 0, i32 1, i64 2, i32 1
  %3 = load i32, i32* %2, align 4
  ret i32 %3
}
```

Figure 5.11: LLVM IR of Figure 5.10

Bounds-narrowing sites can be static or dynamic. At static bounds-narrowing sites, the location within the parent object to which the bounds will be narrowed is known at compile time. All indices within the GEP instruction are constants, as is the case in Figure 5.10. At dynamic bounds-narrowing sites, on the other hand, one or more indices of the GEP instruction are variables, causing the location to only be determined at run time. A variable index can only occur when indexing an array element, not when accessing the field of a structure. As an example, the bounds-narrowing site in Figure 5.11 would be dynamic if the second index, determining the index to the array, was not fixed to 2 but given by a variable instead.

As obtaining the information about which array field is accessed at runtime would add substantial complexity to our

[2] An exception to this are pointer manipulations performed on a pointer after explicitly casting it to another primitive data type in the C source code. However, we can assume that this rarely happens and therefore has a negligible impact on the applicability of our approach.

approach, and we observed that dynamic bounds-narrowing sites typically constitute a relatively small share of bounds-narrowing sites found in real-world programs[3], we decided not explicitly to account for them in our approach.

A further type of GEP instruction that we entirely disregard in our design to reduce complexity are bounds-shifting instructions - GEP instructions in which the first index is not zero. Such instructions use the argument pointer to calculate the pointer to a sibling element in the same array, thereby violating our assumption that the bounds of a given pointer can only remain the same or become narrower when the pointer is modified.

5.8.2. ASM Level

Bounds narrowing on a machine code level can occur in two forms.
Emitting bounds-narrowing instructions take a pointer to a composite object and produce a pointer to a child object, save it in a register or memory and thereby make it available for use by other instructions.
Internal bounds-narrowing instructions, on the other hand, also take a pointer to a composite object and transform it into a pointer to a child object, but only use this pointer themselves. As such, the pointer ceases to exist after the instruction has been executed and cannot be used by other instructions. In practice, this commonly occurs in the computation of the address at which memory is to be accessed according to one of the instruction's operands.

Contrary to the LLVM IR, bounds-narrowing in compiled code can occur in more than one type of instruction. We identify three groups of AMD64 instructions that can represent bounds-narrowing sites in compiled code based on our observations.
The first such group are `add` instructions. These are frequently used for transforming a pointer to a structure into a pointer to a field of that structure by adding a fixed value to the pointer, corresponding to the relative offset from the base of the structure to the beginning of the field. Thus, an `add` instruction can be bounds-narrowing if it satisfies both of the following criteria.

- The incremented number is located in a 32-bit or 64-bit general-purpose register. As outlined in Section 5.7, storing any valid memory address will require at least a 32-bit register.
- The value to be incremented by is immediate. As we disregard dynamic bounds-narrowing instructions in our design, the value to be incremented by is fixed at compile time. Although this does not strictly require it to be represented as an immediate operand, any other representation would be less efficient and can therefore be expected not to occur.

The second type of emitting bounds-narrowing instructions are `lea` instructions, which are typically used for creating pointers and, as such, can act as bounds-narrowing instructions. A `lea` instruction can only be bounds-narrowing if its computation results are stored in a 32-bit or 64-bit general-purpose register, for similar reasons as discussed previously.
The third group of instructions that commonly act as emitting bounds-narrowing instructions are those of the `mov` family. While this might come as a surprise at first due to the inability of `mov`-family instructions to manipulate the moved data beyond generic operations such as sign extension, `mov`-family instructions can be emitting bounds-narrowing instructions in two cases. First, if a `mov`-family instruction is a pointer-creating instruction to the child of a composite static variable. Second, if a pointer to the beginning of the first child of a composite object is created from a pointer to the composite object. In the latter case, both the composite object and its child start at the same address, thus making modification of the pointer unnecessary. Again, a `mov`-family instruction must generally write a 32-bit or 64-bit value to its destination to qualify as an emitting bounds-narrowing instruction.

In principle, any instruction that reads a 32-bit or 64-bit value from a register can be an internal bounds-narrowing instruction. However, as we are only concerned about bounds-narrowing to perform bounds-checking for writes, and we can be sure that the pointer resulting from an internal bounds-narrowing address computation will only be used within this instruction, we can ignore any instruction that is not a write. Accordingly, the only internal bounds-narrowing instructions we need to consider are those of the `mov` family.
Note that emitting and internal bounds-narrowing instructions are not mutually exclusive. For example, a `mov` instruction that writes a pointer referencing the first child of a structure to another structure, calculating the corresponding address within the destination operand, is both an internal and an emitting bounds-narrowing instruction.

We again leverage line number debug information as described in Section 5.6 to match bounds-narrowing instructions in the IR to their counterparts in the compiled code. We outline our approach for matching in Chapter 6.

[3] In fact, dynamic bounds-narrowing sites never occur when arrays of structures are considered unitary objects.

5.9. Determining Intended Pointee Objects

Upon encountering a pointer-creating or bounds-narrowing instruction, the resulting pointer of which may be used for a dependent write immediately or later in the program, determining the intended pointee object (IPO) is vital for performing immediate write checking or tainting the pointer accordingly. We use detailed information about the memory layout, extracted from the DWARF debug information as outlined in Section 5.5, as well as the targets of bounds-narrowing instructions to facilitate this.

In general, four different types of locations exist at which we need to determine the intended pointee object of a pointer at run time, corresponding to the rows in Table 5.3.
The first type of location are dependent writes that are at the same time bounds-narrowing instructions and thereby only permitted to write to a specific child of the object specified by their obtained pointer's taint.
The second type of instructions are those identified as bounds-narrowing instructions by our matching approach that also satisfy our criteria for pointer-creating instructions. In practice, `lea` is the only instruction for which this can occur.
The third and fourth types are instructions that are only bounds-narrowing or pointer-creating.
For each of the four cases, the way in which we handle it is dictated by whether or not the pointer is already tainted.
As indicated in Section 5.5.1, bounds for pointers to fragmented composite objects can never be narrowed. As such, bounds-narrowing as described below only applies to non-fragmented objects.

	Pointer is tainted	**Pointer is not tainted**
BNI + Dep. write	Narrow pointer bounds according to target and use immediately for checking the write	Previously missed pointer tainting, nothing we can do
BNI + PCI	Not actually a PCI, treat like BNI	Use bounds-narrowing target and memory layout to determine IPO and taint the pointer
BNI	Narrow pointer bounds according to target and re-taint	Previously missed pointer tainting, hence treat like BNI + PCI
PCI	Is actually a BNI but was not matched, can only ignore it and leave the taint as is	Find IPO using only memory layout and taint the pointer

Table 5.3: Approaches for determining the intended pointee object of a pointer in different scenarios

Bounds-Narrowing Dependent Write At a dependent write that, at the same time, serves as a bounds-narrowing instruction, an untainted pointer is a clear sign that we missed the pointer creation earlier. As there is no way for us to reasonably infer the intended pointee object at this point, the dependent write will inevitably remain unchecked. However, if the pointer does have a taint, we can easily determine the new intended pointee object from the pointer taint in combination with the bounds-narrowing target and check the write's validity against it.

Bounds-Narrowing Pointer-Creating Instruction If a bounds-narrowing instruction is simultaneously a pointer-creating instruction, our action depends on whether or not the resulting pointer is already tainted.
An already tainted pointer indicates that this instruction was falsely identified as a pointer-creating instruction and, in reality, only serves to transform a pointer to a more narrow pointer. Therefore, we can handle this case as if the instruction was only bounds-narrowing.
If the resulting pointer is not yet tainted, this is a strong indicator of the instruction indeed being a pointer-creating instruction that creates a pointer to a child of a composite object. We can validate this by inspecting the object pointed to by the resulting pointer. If this is a composite object of the same type specified by the bounds-narrowing target, we taint the resulting pointer accordingly. Otherwise, the pointer is either already out-of-bounds, or the instruction was incorrectly identified as bounds-narrowing, and we cannot reasonably taint.

Bounds-Narrowing Instruction The treatment of a bounds-narrowing instruction that is not pointer-creating again depends on whether the resulting pointer is already tainted or not.
If the pointer is tainted, we determine the intended destination object's type from the taint and verify that it is the same as the type specified by the bounds-narrowing target. If this is the case, we determine the new intended destination object using the bounds-narrowing target and re-taint the pointer. If the types do not match, the pointer has either been tainted incorrectly before, or the bounds-narrowing instruction has been incorrectly identified, preventing us from performing bounds-narrowing.
If the pointer is not tainted, this indicates that a pointer was created without being tainted immediately. We describe possible reasons for this in Section 5.7. However, the availability of bounds-narrowing information here allows us to taint it by treating it as if it was a bounds-narrowing pointer-creating instruction.

Pointer-Creating Instruction If the pointer-creating instruction is not a bounds-narrowing instruction at the same time and, as such, does not have a bounds-narrowing target associated with it, we have to infer the intended destination object solely from the pointer itself and the extracted memory layout.
Although the absence of a bounds-narrowing instruction here indicates that the pointer is freshly created from an immediate address or a stack frame boundary register and is therefore not tainted, we might encounter a tainted pointer if the instruction does act as a bounds-narrowing instruction but was not identified as such. In this case, we could theoretically attempt to infer the intended pointee object from the pointer in combination with the extracted memory layout and re-taint accordingly. However, as the lack of information on the bounds-narrowing target introduces an increased risk of introducing faulty bounds, we refrain from this and leave the taint unchanged. If the resulting pointer is not tainted, we can encounter the following three different cases.

Points to unitary object: This is the simplest case. The unitary object pointed to is the intended pointee object.

Points to composite object: In this case, it is initially unclear whether the intended pointee object is the composite object itself or one of its (nested) children. However, as any pointer-creating instruction that creates a pointer to the child of a composite object would at the same time be a bounds-narrowing instruction, we can infer from the absence of a bounds-narrowing target that the entire composite object is the intended destination object.

Points to unused space: This case occurs when a pointer to an object not listed in the DWARF debug information is created. An example of such an object is a spilled register value. In this case, there is no way of reliably determining the intended pointee object. Thus, we cannot taint the pointer accordingly.

6 Implementation

The previous chapter provided an in-depth description of our design for a non-invasive OOB write characterization approach. To demonstrate the practicability of this design and evaluate its performance, we created a prototype implementation of our approach. This chapter provides a description of this implementation, outlining its structure and solutions for the different implementation challenges.

We first provide a high-level overview of our implementation, describing the general architecture and functions of the different components. Afterwards, we describe each of our implementation's three components in detail, along with the challenges that this component tackles.

6.1. Overview

We implemented a prototype of our non-invasive OOB write characterization approach on top of the S2E platform, supported by static analysis components in the form of a pass for the LLVM compiler framework, as well as a component to extract relevant information from compiled binaries. Our implementation comprises 2700 SLOC of C++ and 1900 SLOC of Python code and is made publicly available on GitHub[1].

A high-level schematic diagram of our implementation is presented in Figure 6.1. The pipeline takes as input the source code of the program under test, as well as a proof-of-concept (PoC) input to the program under test that is suspected of causing an OOB write. It produces an output file summarizing the effects of all discovered OOB writes during execution on source code-level objects in an easily interpretable fashion.

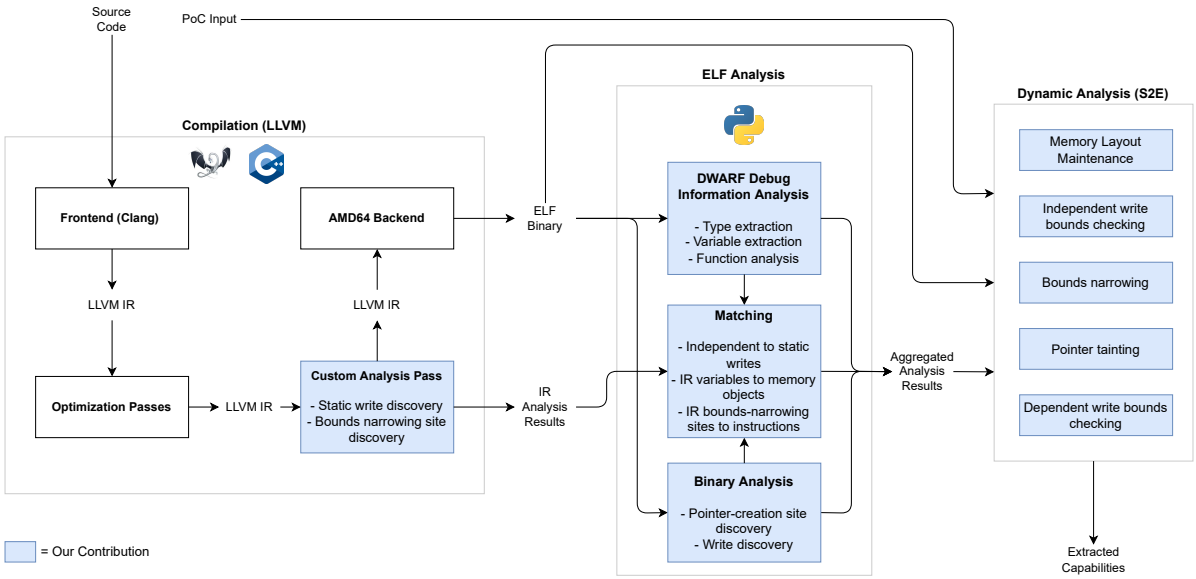


Figure 6.1: Schematic diagram of the pipeline implementing our design

[1] https://github.com/ethoxx/noninvasive-oobw-characterization

The first stage compiles the program under test and yields the compiled executable together with the results of the LLVM IR analysis. It largely follows the standard procedure for compiling a program with the LLVM infrastructure, using the Clang frontend to transform the C source code to the LLVM intermediate representation. The LLVM IR is then optimized through several optimization passes, the exact type and arrangement of which is determined by the optimization level. As the last step before the LLVM IR is transformed to another representation and finally compiled to machine code and linked into an ELF executable, our analysis pass is applied to the LLVM IR to extract high-level semantic information on static writes and bounds-narrowing sites. This information is combined and output as the result of the compilation stage, next to the ELF executable. It is important to note that our analysis pass does not modify the LLVM IR but merely analyzes it. As such, the resulting ELF binary will be the same as if our analysis pass was not applied[2]. This allows for integrating it into existing build processes for production builds.

The second stage in our pipeline takes as input the compiled ELF binary and the IR analysis results from the compilation stage. It analyzes the DWARF debug information to extract variables, types, as well as information on functions and analyzes the compiled code to discover pointer-creating instructions, dependent writes, and independent writes. This information is combined with the IR analysis results to identify intended destination objects of independent writes and the exact instructions and targets of bounds-narrowing instructions. Finally, all results are aggregated and output for ingestion by the next stage.

Our pipeline's third and final stage serves to execute the program under test and closely monitor it during execution, implementing the procedure described in Algorithm 1 to detect OOB writes. It performs bounds checking for independent and dependent writes, the latter of which relies on pointer tainting and bounds narrowing. Furthermore, the program's memory layout is constantly tracked to facilitate bounds checking and the identification of affected objects.

6.2. LLVM IR Analysis

We implemented all analyses of the LLVM IR in the form of an LLVM module pass that is run on the IR during compilation. We further outline how this LLVM pass finds and extracts necessary information about static writes and bounds-narrowing sites, followed by a discussion of the possibilities for integrating the analysis pass into a build process.
The analysis results of this stage are emitted as a JSON-formatted file, following the format specified in Table A.1.

As outlined in Chapter 5, matching static writes and bounds-narrowing sites to their counterparts in the compiled code relies on line number debug information. However, we observed a substantial decline in the quality of this debug information with increasing compiler optimizations, which frequently causes multiple instructions in the compiled code to share the same line number information and occasionally causes the loss of all line number information for parts of the program. The consequence of this is a substantially reduced effectiveness of our matching in the ELF analysis stage and thereby a decreased effectiveness of bounds checking at run time.
To alleviate this issue, we extended our analysis pass to introduce synthetic line number debug information to static writes and bounds-narrowing sites with line number information colliding with other instructions or missing any line number information altogether. Note that we do not attempt to make the added debug information credible with regard to source code locations - it serves exclusively to distinguish instructions. As debug information does not affect the code generated by the compiler, this does not violate our goal of non-invasiveness.

6.2.1. Extracting Static Writes

For each instance of an LLVM IR instruction that we identified as the potential source of a write in the compiled code, i.e., `store`, `llvm.memcpy`, and `llvm.memset`, we initially obtain the identifier specifying the destination of the write. We then traverse the declaration chain upwards as described in Section 5.6, maintaining a vector of structure and array indices that specify the bounds narrowing operations encountered in the declaration chain.

Reaching a terminating instruction indicating a dynamic write causes the write to be discarded from our analysis results since a dynamic write can never be translated to an independent write, as described in Section 5.4. On the other hand, in the case of a terminating instruction indicating a static write, we retain the source code-defined name of the corresponding variable for inclusion in the analysis results, together with the line, column, and file described in the debug information attached to the write, as well as the collected bounds narrowing indices.

[2] The only modification that our analysis pass performs on the LLVM IR concerns debug information, which therefore also affects debug information in the final ELF binary. However, this only affects the debug sections in the binary, which are typically not included in production builds. The actual code, on the other hand, remains unchanged.

6.2.2. Extracting Bounds-Narrowing Sites

Bounds narrowing sites in the IR generally occur as `getelementptr` (GEP) instructions or operators. While GEP instructions are independent instructions that produce a new SSA variable, GEP operators can only exist as the operand of another instruction. Therefore, they cannot have debug information directly attached to them. Furthermore, the result of a GEP operator can only be used by the containing instruction.

To discover bounds narrowing sites in the IR, we inspect every instruction and check whether it is either a GEP instruction or an instruction containing a (nested) GEP operator. If so, we collect the bounds narrowing indices describing the bounds narrowing target from the GEP if all of the following conditions hold:

1. There is only a single GEP associated with this instruction. If there are multiple, all of them will have the same line number debug information in the compiled binary, making it impossible to distinguish and match them. We record the number of times this occurs in the pass statistics.
2. The aggregate object to which the GEP takes a pointer and converts it into a pointer to a child object is a composite object according to our definition in Section 5.1. Thus, we only consider GEPs that operate on a type containing at least one structure. Any other type would correspond to a unitary object within which no OOB writes are possible according to our definition.
3. The GEP is a static bounds-narrowing instruction. If any index of the GEP that specifies an element or field to which narrowing is performed is not a constant, the bounds narrowing instruction is dynamic and the child object to which it creates a pointer is only determined at run time, as outlined in Section 5.8.1. We record the number of times this occurs in the pass statistics.
4. The GEP is not a bounds-shifting instruction, i.e., the first index of the GEP instruction is zero, and the interval describing the bounds of the resulting pointee is therefore fully contained in the bounds of the input pointee.
5. The result of the GEP is potentially used for a dynamic write. We inspect each direct and indirect usage of the GEP's resulting variable and check whether it is possibly used for determining the destination of a dynamic write. If this is impossible for all uses, the GEP can safely be ignored.

If all conditions hold, we retain the line, column, and file described in the debug information attached to the instruction, as well as the bounds-narrowing indices and a mnemonic string describing the type to which the bounds-narrowing instruction accepts a pointer. This mnemonic is used during the dynamic analysis to compare the expected object type at the bounds narrowing instruction with the object type indicated by the pointer's taint. If the result of the GEP is only used at a single instruction, we additionally record the debug line information attached to this instruction.

6.2.3. Build Process Integration

Following from our design's reliance on analysis of the LLVM IR, the implementation of our LLVM analysis is generally only compatible with build processes using Clang for compilation. However, as production builds commonly do not emit LLVM IR that we can run our analysis pass on, slight modifications to the build process are required.

While there are several ways to integrate our IR analysis into the build process, some of which prevent the addition of synthetic debug information, we found the approach of running our analysis pass in-line during compilation to be most suitable. For simple build processes not relying on build systems such as CMake, the pass can be inserted by providing additional arguments to Clang. Build processes relying on CMake require a single additional line in the CMake configuration file.

As the usability of our analysis pass results decreases with an increasing number of transformations between the analysis and generation of the final code, we strive to run the pass as late as possible during compilation. This is achieved by instructing Clang to schedule the pass at the `EP_OptimizerLast` extension point, which causes it to be run as the last pass before the IR is transformed into another representation. Note that this assumes the absence of link-time optimizations, which would require the entry point `EP_FullLinkTimeOptimizationLast` to achieve similar goals.

Besides relying on the LLVM IR analysis, the availability of full debug information is also vital for our approach. We can easily instruct the compiler to provide full debug information with the `-g3` command line argument. As the effects of attaching debug information to a binary are limited to including additional sections in the ELF file, debug information can easily be stripped from the binary to remove any trace of the modified build process.

6.3. ELF Analysis

The second stage of our approach's implementation - the analysis of the ELF file generated by the linker - consists of three major components that we describe in this section.

In the first step of the ELF analysis implementation, the ELF file is loaded, and the contained DWARF debug information is analyzed as outlined below. Afterwards, the JSON file containing the results of the LLVM analysis pass is parsed and combined with the information extracted from DWARF, followed by an analysis of the machine code. Finally, the information gained from the IR, DWARF, and machine code analysis is combined to facilitate matching independent writes and bounds-narrowing instructions.

As in the LLVM IR analysis implementation, the results of the ELF analysis are transferred using a JSON file.

6.3.1. Debug Information Analysis

Analyzing the DWARF debug information attached to the ELF file provides us with an abundance of information vital for keeping track of the program's memory layout and matching static writes and bounds-narrowing instructions to their counterparts in the compiled code. Our implementation for analyzing the DWARF debug information makes heavy use of the *pyelftools* library by Eli Bendersky [55], which provides an extensive API to parse and examine DWARF information.

To extract the information relevant to us, we visit every DIE and check whether it is of an interesting type. DIEs with the `DW_TAG_subprogram` tag describe a function in the program under test and contain relevant information such as the function name, the start and end of the function in the compiled code, as well as the frame base register. DIEs with the `DW_TAG_variable`, `DW_TAG_formal_parameter`, and `DW_TAG_constant` tags represent source code-defined entities that potentially have a representation in memory. Their attributes describe the object's source code-defined name, type, and location over its entire lifetime.
If the object is never located in memory over its lifetime, we can ignore it. Otherwise, we inspect its fragmentation and internally record each fragment and lifetime interval for inclusion in the analysis results. Furthermore, we inspect its type and classify it as a unitary, array, or structure type according to our definition in Section 5.1. In the case of an array type, we record the number of elements and the type of the elements. For a structure type, we record the name and type of each field, together with its offset from the start of the structure.

If a restriction on the lifetime of a stack memory object is imposed by the lifetime of an ancestor DIE, such as a `DW_TAG_inlined_subroutine` or a `DW_TAG_lexical_block` DIE, we incorporate this restricted lifetime in our record of the memory object. However, if the lifetime of an ancestor DIE consists of multiple non-adjacent intervals, we only limit the lifetime of the memory object to the interval from the start of the earliest to the end of the latest interval to decrease implementation complexity. Occasionally, this simplification causes our records to show multiple memory objects located at the same position and time.
To combat the complications that would arise from this during dynamic analysis and to detect and eliminate variables that Clang merges during compilation with aggressive optimizations, we inspect all extracted stack memory objects for overlaps with other memory objects after finishing the analysis of DWARF debugging information. If multiple objects with identical spatial dimensions are located at the same position simultaneously, we merge them. The requirement of merged objects having the same spatial position prevents this from inhibiting our detection performance.
Nevertheless, not all memory objects recorded as overlapping due to our simplifying assumption also have the same spatial dimensions and position. Thus, some objects cannot be merged, making it impossible to determine the intended pointee of a pointer that points to an address at which they overlap during dynamic analysis. However, our observations show that even in heavily optimized and inlined programs, the share of overlapping stack objects typically does not exceed 3%. Thus, it is unlikely to decrease our implementation's performance substantially.

All information resulting from the analysis of DWARF debug information is presented in Appendix A.

6.3.2. Code Analysis

Analyzing the machine code generated by the compiler is essential for identifying pointer-creating instructions, independent writes, and for matching static writes and IR bounds-narrowing sites to individual ASM instructions. Furthermore, it provides various types of metadata for our dynamic analysis implementation. Our implementation for analyzing the compiled code of an ELF file relies on the Capstone Engine [56] for disassembling and examining machine code.

To identify the pointer-creating instructions in the program under test, we inspect each instruction and check whether it satisfies our definition of a pointer-creating instruction described in detail in Section 5.7. If so, we record the instruction's address and the register to which it writes the created pointer for inclusion in our analysis results.

We identify independent writes in the program under test according to the criteria described in Section 5.4. Non-innocuous independent writes are internally recorded to match them to their corresponding static write. The addresses of innocuous independent writes, as well as ordinary innocuous writes such as `push` and `call`, are

included in our analysis results to facilitate distinguishing between innocuous writes that we deliberately excluded from checking and dependent writes remaining unchecked due to missed pointer tainting.

Various operations performed in our dynamic analysis implementation rely on the knowledge of the location of different program sections. Therefore, we extract all sections' names, starting addresses, sizes, and permissions from the program header.

To facilitate hooking into external library function calls, which we further describe in Section 6.4.6, we generate a mapping from addresses in the procedure linkage table, which are called in the program under test to invoke the external function call, to function names. Such a mapping is not directly available in an ELF file but can be generated via the `.rela.plt` section, which contains a mapping of shared library function names to their corresponding pointer location in the global offset table's procedure linkage table (`.got.plt`). Therefore, we can use the `call` operand address to look up the pointer to the function's `.got.plt` entry in the `.plt`, allowing us to match called addresses to function names.

6.3.3. Matching

Matching independent writes to memory objects and bounds-narrowing instructions to their corresponding ASM instruction is crucial for bounds-checking independent writes and detecting dependent intra-object OOB writes.

Independent Writes For each non-innocuous independent write previously discovered, we try to find the corresponding static write by searching for a static write whose line, column, and file name extracted during the IR analysis matches the line, column, and file name specified by the DWARF line number debug information for this instruction's address. However, under any of the following conditions, we cannot identify a unique match for the independent write and are forced to ignore it, leaving it unchecked during dynamic analysis. We evaluate the frequency with which this occurs in Chapter 7.

- The independent write has no line number debug information
- There are other dependent or independent writes with the same line number debug information
- There exists no static write with the same line number debug information
- There exists more than one static write with the same line number debug information

When the static write of an independent write has been successfully determined, it remains to identify the memory object described by the DWARF information that the static write is intended to write to. This is easily done by matching on the source code-defined object name, which is present both in the IR analysis results and the DWARF information. If the object is composite and the IR analysis revealed that the write is intended to affect only a child of the object, we identify this child from the bounds-narrowing indices specified by the IR analysis results. Note that this is not possible if the object is fragmented, as discussed in Section 5.5.1.

Bounds-narrowing Instructions Matching bounds-narrowing instructions discovered in the IR to their counterparts in the machine code is essential for providing the latter with information on the bounds-narrowing target and enabling us to narrow the bounds to the correct child of the composite object during dynamic analysis.

To reduce the complexity of our implementation, we do not attempt to match to emitting bounds-narrowing instructions that write the resulting pointer directly to memory. On the one hand, this decision is motivated by the fact that this prevents us from having to taint pointers located in memory. On the other hand, it also eliminates the possibility of an instruction having to perform bounds-narrowing twice - for the emitted pointer and for the address calculation to which the pointer is written. As we have not observed any instruction affected by this self-imposed limitation in practice, it is unlikely to affect our ability to detect intra-object OOB writes.
Similarly, we disregard instructions of the `cmov` family as bounds-narrowing instructions to reduce implementation complexity. Again, we rarely observed the occurrence of such instructions in practice.

To find the machine code instruction corresponding to a bounds-narrowing instruction discovered during the IR analysis, we first identify all instructions with the same line number debug information - line number, column number, and file name. If there are no such instructions, which frequently occurs when the bounds-narrowing is performed within a dynamic write, and if we found only a single user of the narrowed pointer during the IR analysis, we also consider instructions matching the debug line information of the user. Afterwards, we remove all instructions that, according to our analysis in Section 5.8, are not a potential candidate for a bounds-narrowing instruction with the respective target. Now, if there is only one potential candidate, we consider it the matching instruction. If there is more than one candidate instruction, we can identify a matching instruction if the following conditions are satisfied.

1. The n candidate instructions are sequential; there are no instructions between them that were previously removed.
2. There exists exactly one general-purpose register (GPR) r that is modified by the first $n-1$ instructions. Each of the first $n-1$ instructions modifies either the GPR r or no GPR.
3. None but the last instruction modifies memory.

If these conditions are satisfied, we can pick the last candidate instruction that writes to r as the bounds-narrowing instruction. The motivation for this selection procedure are instruction patterns frequently found in unoptimized code, where a pointer is loaded from memory into a register, modified in the register, and finally written back to memory. As such patterns cause multiple instructions to have the same line number debug information, such a selection procedure is necessary.

6.4. Dynamic Analysis

We implemented the dynamic analysis part of our approach in the form of three plugins for the S2E platform [35].

The `OOBTracker` plugin implements the core logic of our approach. It monitors the program under test for writes, pointer-creation, and bounds-narrowing and performs pointer tainting, write checking, and library function call interception.
The `MemoryTracker` plugin implements memory layout tracking and provides the `OOBTracker` with the means to obtain information such as the object located at a specific address. Furthermore, it ingests and maintains all information provided by the ELF analysis stage, allowing the `OOBTracker` to query it for information such as the function starting at an address, as well as whether an instruction constitutes a pointer-creating or bounds-narrowing instruction.
The `OOBAnalyzer` plugin records all discovered OOB writes, aggregates them, and outputs them in an easily interpretable fashion upon the program's termination.

We further outline our approach to solving the core implementation challenges, as implemented in the three plugins.

6.4.1. Memory Layout Tracking

Tracking the position of objects in memory at run time is facilitated by the information previously extracted from the DWARF debug information. Objects at static locations, i.e., in the `.bss` and `.data` sections, are simple to track. On the stack, however, the utilization of a given memory address depends on the current call stack and the value of the stack frame boundary registers during the execution of the function in whose stack frame the address is located.

We therefore maintain an internal model of the current call stack that is updated every time a function is called via a `call` instruction or returns via a `ret` instruction, as described in Section 5.5. When seeking to identify the current utilization of a memory address, we can use this to determine the stack frame within which the address is located and the corresponding function. From the value of the stack frame boundary registers during execution, together with the relative locations of objects within the stack frame which we extracted from DWARF during the ELF analysis, we can finally determine the object by which the address is utilized.

6.4.2. Pointer Tracking

The necessity of tracking pointers through the program in a fashion similar to dynamic taint analysis is a significant motivation for implementing our approach on top of an emulation layer, as it is provided by QEMU in S2E.
Although S2E does not explicitly support performing dynamic taint analysis on the program under test, its symbolic execution capabilities can be repurposed to facilitate something akin to it. When making a piece of data symbolic by replacing it with a corresponding KLEE expression, the name of the symbolic expression can be used as a unique identifier, facilitating its use as the key in a lookup table that matches expression names to pointee objects. As any data derived from a symbolic value is also symbolic and accompanied by a symbolic constraint in the form of a KLEE expression, it is possible to find the symbolic values from which a symbolic value was created, thereby effectively allowing for performing taint analysis.
Thus, to taint a pointer, we make it symbolic with a globally unique name and record the name together with information on the intended pointee object in a table. Each name of a symbolic value representing a pointer contains a fixed prefix to allow for unambiguously identifying it as a pointer. To identify the intended pointee object from a symbolic pointer, we obtain the name of the symbolic value constituting the base of the symbolic constraint that accompanies the symbolic pointer and resolve it to an object via a lookup table.

In some cases, however, an address might be derived from more than one pointer. This occurs when pointer arithmetic is used on multiple pointers either explicitly by the user or implicitly by the compiler. An example of this is calculating a relative memory address offset by subtracting one pointer from another. This causes the resulting value to become doubly tainted, although it should inherit the intended destination object from neither of the original pointers.
We solve this issue by inspecting the symbolic constraints of pointers when identifying the taint. For each subtraction

expression with a symbolic minuend x or symbolic subtrahend y, we check whether $|x - y| < 0x300000$. If so, we conclude that the pointers annihilate each other to compute a relative offset and therefore consider the resulting value untainted. The choice for the fixed value `0x300000` is motivated by the fact that the virtual address space interval `[0,0x400000)` typically remains unmapped for processes on Linux systems. The difference of `0x100000` is merely a precaution to prevent easily avoidable false positives when using this heuristic.
A simplified example of a symbolic constraint to which this heuristic applies is presented in Figure 6.2, representing the computation `0x1969607 + 0x7ffdbf3c4778 - 0x7ffdbf3c4290`. Here, a relative distance between two stack addresses is used as an offset to access a location on the heap. Clearly, the intended destination object here is given by neither of the two tainted pointers.

```
(Add w64 0x1969607
         (Sub w64 (ReadLSB w64 0x0 v16_memobj_0x7ffdbf3c4778_16__16)
                  (ReadLSB w64 0x0 v15_memobj_0x7ffdbf3c4290_15__15)))
```

Figure 6.2: Example of symbolic address constraint derived from multiple pointers

6.4.3. Independent Write Checking

Although the S2E API provides the event `onConcreteDataMemoryAccess`, with which subscribers are informed upon the program under test accessing memory using a concrete address, this event can, by default, not be used for checking the validity of static writes. This is because S2E optimizes the execution of translation blocks (TLBs) that do not touch any symbolic data by only updating the content of the instruction pointer register within S2E's internal execution state after executing the entire TLB, not after individual instructions. Consequentially, if the instruction pointer is obtained upon a concrete memory access event, it will always be inaccurate unless the memory-accessing instruction is the very first instruction of the TLB. This prevents us from checking the bounds of the independent write, as we cannot identify the instruction from which the write originates and, as such, cannot determine the intended destination object.
We solve this problem by extending S2E to expose a new event that is emitted each time an instruction within a TLB is translated and allows subscribers to inject an internal instruction for synchronizing registers with the execution state. By ensuring that this synchronization instruction is injected after every instruction located in the `.text` section of our program under test, we can leverage the `onConcreteDataMemoryAccess` event for bounds-checking independent writes.
When the plugin is notified of the event, we obtain the destination address of the write together with the number of bytes written and check that all written bytes are within the bounds of the independent write's intended destination object. If any written byte is out-of-bounds, we report it to the `OOBAnalyzer` plugin for further handling.

6.4.4. Dependent Write Checking

A consequence of tainting pointers by making them symbolic is that any dependent write to an address given by a pointer that previously has been successfully tainted will write to a symbolic memory address. By subscribing to the event `onBeforeSymbolicDataMemoryAccess` exposed by the S2E API, we can intercept each such write and perform bounds checking.
When our plugin is notified of a write to a symbolic memory address, we obtain the symbolic address and identify the intended pointee object as described in Section 6.4.2. Then, we obtain the concrete address from the symbolic value and the size of the write in bytes and check whether each byte to be written is contained in the interval of the intended destination object. Then, again, we report any OOB write to the `OOBAnalyzer` plugin for further handling.

6.4.5. Pointer Creation and Bounds Narrowing

We handle pointer-creating and bounds-narrowing instructions by subscribing to the `onTranslateInstructionEnd` event that is emitted for each instruction during the translation of a TLB. Upon invocation, we check whether the instruction at this address was identified as a pointer-creating or bounds-narrowing instruction during our ELF analysis stage. If so, we inject a callback to be invoked upon executing the instruction.
When the callback is invoked, we handle the instruction as described in Section 5.9, determining the new intended pointee object based on the taintedness of the pointer, the instruction type, the bounds-narrowing information, as well as the type of the old intended pointee object. If this succeeds, we taint the pointer accordingly and continue execution.

For successfully tainting a pointer in our handler, it is essential to ensure that S2E is in symbolic mode before executing the instruction. If the execution mode was switched to symbolic in the handler to create new symbolic values, this would prevent the instruction from being re-executed, leaving the pointer untainted. We solve this by registering an event handler for `onTranslateInstructionStart` that forces S2E to switch to symbolic mode if the next instruction is pointer-creating or bounds-narrowing.

As previously indicated, we refrain from tainting pointers resulting from bounds-narrowing instructions that write the pointer directly to memory to reduce implementation complexity. This also holds for pointer-creating instructions

that write directly to memory. According to our observations, this affects less than one percent of pointer-creating instructions in typical real-world programs. Thus, we do not expect this to substantially impact our implementation's performance.

As we observed that pointers to objects are occasionally created before the object's lifetime starts according to the DWARF debug information, we opted to implement a small temporal tolerance window when determining the intended pointee of a pointer from the memory layout alone. Thus, if a pointer to an address is created at which we did not record any objects at this instruction address, we also consider objects whose lifetime starts at most 256 bytes of instructions later. This substantially improves the success rate of determining intended pointees without introducing any noticeable side effects.

6.4.6. Library Function Call Interception

An important advantage of our implementation using the S2E platform for tainting pointers is the compatibility with external library code. While other pointer-based approaches described in Section 3.2 typically require wrappers for external function calls to achieve compatibility, this is not the case for our approach. In addition to the compatibility, our approach even allows for detecting OOB writes in such external library functions if the pointer from which the OOB write originates was created and tainted in a function within our scope. As such, we can detect OOB writes within functions such as `memcpy`.

However, any library function to which a tainted pointer is passed will execute mainly in KLEE mode, caused by the symbolic value. Combined with the internal complexity of some C standard library functions, this causes a substantial slowdown of execution in library function calls. To alleviate this, we intercept calls to some C standard library functions and concretize their arguments in the general-purpose registers described in Section 2.2.1. As these general-purpose registers are caller-saved, we can be certain that concretization does not affect bounds-checking in the calling function after the library function returns. However, as the concretization of function arguments prevents the detection of OOB writes within the library function, we perform premature bounds checking by obtaining the arguments and validating that the operation performed by the library function will be in-bounds. This is facilitated by the well-defined interface of C standard library functions, which allows foreseeing the exact effects of a function call.

However, such premature bounds-checking is not possible for all library functions, as the logic determining affected memory ranges in some cases is too complex to simulate in a wrapper or depends on information that is not readily available. Therefore, for some functions, we concretize only arguments that are not pointers used for writing within the function. An overview of the functions we intercept, together with the information on whether we perform premature bounds-checking and argument concretization, is presented in Table 6.1.

Libc function	strcpy	strncpy	memcpy	memset	sprintf	snprintf	strcat	strncat	sscanf	fscanf	printf	fprintf	malloc	free	fopen
Premature bounds-checking?	✓	✓	✓	✓	×	×	✓	✓	×	×	-	-	-	-	-
Argument concretization?	✓	✓	✓	✓	◯	◯	✓	✓	◯	◯	✓	✓	✓	✓	✓

Table 6.1: Libc functions intercepted for premature bounds-checking and argument concretization (✓= yes, ◯= partly, ×= no, -= n.a.)

6.4.7. Collection and Reporting

Presenting discovered OOB writes in a concise and structured manner is essential for making the results easily interpretable and reaching our overall goal of providing human analysts and developers with a way to get a quick overview of a vulnerability's capabilities.

An important decision for a concise presentation of discovered OOB writes is when to consider multiple OOB writes as the same vulnerability, making it necessary to group them. If each OOB write was presented separately, this would cause the clarity of the results to suffer substantially, as vulnerabilities causing an OOB write in a loop would potentially yield hundreds or thousands of entries. Therefore, we opted to consider multiple OOB writes to belong to the same vulnerability and group them if all of the following conditions are satisfied.

1. All OOB writes occur at the same instruction within the program.
2. The call stack at the time of the OOB write is the same for all of them. We consider two call stacks equal if both correspond to the same sequence of function calls with equal call locations, and each two stack frames are located at the same position.

For every vulnerability, we merge intervals of memory addresses overwritten during each individual OOB write. For each interval, we identify the affected objects and present their name, size, and the first and last byte overwritten within them. Although not source code-level objects, we also display the saved instruction pointer and the saved stack frame base pointer if they are overwritten. This is facilitated by their predictable position in each stack frame.

To make the results easily interpretable for humans and other tools, we present them in a JSON-formatted fashion. An example, created by running the PoC for CVE-2017-9047 on the xmllint tool of libxml, is presented in Figure 6.3. Here, it is immediately visible that two different OOB writes occur during program execution. Although they occur at different instructions, both originate from the same function and can presumably be attributed to the same root cause. While the OOB write at the first instruction overwrites a total of 685 bytes, most of them in the `list` object, the OOB write at the second instruction overwrites only the two bytes located immediately after the end of the first OOB write.


```
[
  {
    "function": "xmlSnprintfElementContent",
    "instruction": "0x48a2c9",
    "pruned_call_stack": false,
    "affected_ranges": [
      {
        "overwrite_lower": "0x7fff7f6fdbe8",
        "overwrite_upper": "0x7fff7f6fde95",
        "affected_stack_frames": [
          {
            "function": "xmlValidateOneElement",
            "stack_frame_bottom": "0x7fff7f6fefff",
            "stack_frame_top": "0x7fff7f6fc860",
            "overwrite_lower_rel": "RBP-5128",
            "overwrite_upper_rel": "RBP-4444",
            "affected_objects": [
              {
                "name": "<unknown/padding>",
                "startAddr": "RBP-5128",
                "size": 8,
                "first_overwritten_byte": 0,
                "last_overwritten_byte": 7
              },
              {
                "name": "list",
                "startAddr": "RBP-5120",
                "size": 5000,
                "first_overwritten_byte": 0,
                "last_overwritten_byte": 676
              }]}]}]},
  {
    "function": "xmlSnprintfElementContent",
    "instruction": "0x48a3c7",
    "pruned_call_stack": false,
    "affected_ranges": [
      {
        "overwrite_lower": "0x7fff7f6fde96",
        "overwrite_upper": "0x7fff7f6fde96",
        "affected_stack_frames": [
          {
            "function": "xmlValidateOneElement",
            "stack_frame_bottom": "0x7fff7f6fefff",
            "stack_frame_top": "0x7fff7f6fc860",
            "overwrite_lower_rel": "RBP-4443",
            "overwrite_upper_rel": "RBP-4443",
            "affected_objects": [
              {
                "name": "list",
                "startAddr": "RBP-5120",
                "size": 5000,
                "first_overwritten_byte": 677,
                "last_overwritten_byte": 677
              }]}]}]
```


Figure 6.3: Example of analysis output, resulting from running the PoC for CVE-2017-9047 on xmllint compiled with -O3

7 Evaluation

In this chapter, we evaluate our approach's performance from several different perspectives. First, we provide an overview of our evaluation approach, outlining the experiments and their goals. Afterwards, we describe the detailed setup of each of our four experiments and present the results.

7.1. Overview

We now provide an overview of our evaluation approach, outlining the types of experiments we conduct, how we compare our approach to existing state-of-the-art sanitizers, and how we evaluate the effects of compiler optimizations on our approach's performance.

7.1.1. Experiments

We evaluate our approach's performance in four aspects using the following experiments.

Dependent write checking We evaluate our approach's ability to detect dependent writes by applying it to a testbed consisting of 122 test cases covering dependent OOB writes in all three monitored memory regions - stack, `.data`, and `.bss`, using different functions and targets. The goal of this is to evaluate the correct detection of OOB writes, i.e., the relationship between true positives and false negatives. As this requires a large number of different spatial memory bugs to evaluate on and can therefore not be realistically performed using real-world vulnerabilities due to their comparatively low density in programs, we resort to a testbed and evaluate under laboratory conditions.

Independent write checking As our procedure for detecting dependent OOB writes significantly differs from that for independent OOB writes, we evaluate it separately on another testbed. This testbed consists of 44 test cases covering independent OOB writes in all three memory regions, using different techniques, contexts, and targets.

Real-world vulnerabilities We also evaluate our approach on a number of real-world vulnerabilities. This shows that our approach is applicable to larger programs with real vulnerabilities and serves to evaluate the prevalence of false positives caused by our approach. The latter cannot be effectively shown with the evaluation on testbeds, as these, by design, contain relatively few writes in total, with a very high density of OOB writes. Here, we additionally present a number of statistics that provide insights into the performance of our approach's internal components.

Performance overhead Although a low performance overhead was not a primary goal in our approach design, we evaluate the performance penalty incurred by our implementation compared to native execution and two state-of-the-art sanitizers.

7.1.2. Comparison to Sanitizers

To put the performance of our OOB write detection approach into context, we perform a comparison to the state-of-the-art sanitizers AddressSanitizer [23] and SoftBound [41] for each of the four evaluation aspects. We use ASan as it is included in Clang 13.0.1 and SoftBound with Clang 3.8.0, the latest publicly available version. To facilitate better comparability between our approach and ASan, we instruct ASan not to terminate upon a detected memory safety violation and continue execution until the program terminates by itself. To achieve similar behavior for SoftBound, we modify its source code not to terminate after detection.

7.1.3. Influence of Optimizations

To investigate the influence that optimizations have on the performance of our approach, we perform each experiment on multiple optimization levels. This serves to answer our third research question and assess the applicability of our approach to binaries built for real-world deployment, which are frequently optimized.

As of version 13.0.1, the Clang compiler features seven different optimization levels. These range from no optimizations to optimizations that aggressively attempt to decrease the program's running time at the cost of increased size and longer compilation time and optimization levels that aim at aggressively reducing the executable size [57].

- `-O0` disables all optimizations, resulting in easily debuggable code that is both slow and comparatively large.
- `-O1` is the default optimization level applied via the optimization flag and performs moderate optimizations to decrease running time and executable size.
- `-O3` performs aggressive optimizations to decrease the program's running time.
- `-Oz` performs aggressive optimizations to reduce the size of the resulting code.

We selected the four optimization levels to cover a large variety of employed optimizations. The omitted level `-O2` performs running time optimizations with an aggressiveness between that of `-O1` and `-O3`, therefore presumably yielding results between those two. Similarly, the omitted level `-Os` performs executable size optimizations with an aggressiveness between `-O0` and `-Oz`. The last omitted optimization level, `-Ofast`, on the other hand, performs running time optimizations beyond `-O3` but generates code that may violate compliance with language standards, thus potentially making it unusable with our approach.

7.2. Dependent Writes

In our approach, the detection of dependent writes, i.e., writes relative to pointers, is facilitated by a pointer tainting mechanism that allows for augmenting pointers to carry additional information through the program. Evaluating its performance is essential as it offers ample room for various issues.

7.2.1. Setup

To measure our approach's ability to detect dependent OOB writes, we use the RIPE testbed created by Wilander *et al.* [16]. This testbed is designed to evaluate the performance of defense mechanisms against buffer overflow exploits. It combines multiple hundred test cases composed of five attack characteristic dimensions in a single program. Although our approach is not aimed at providing defenses against buffer overflows, the fact that all attacks in the testbed exploit an OOB write makes it a suitable testing environment for us. As the original RIPE testbed described in the paper by Wilander *et al.* is incompatible with some traits of Clang's linking procedure for 64-bit executables and can therefore only be compiled to a 32-bit executable, we instead use a 64-bit port of RIPE [58].

Attacks in the RIPE testbed generally make use of a buffer overflow, manipulating a code pointer that is eventually loaded as the instruction pointer to divert control flow and eventually resulting in a shell being spawned. As such, the test cases are classical control data attacks.

Although combining the different options for the five dimensions of parameters can yield more than 800 different test cases, only a subset of these parameters is relevant for us. We now provide an overview of such parameters.

Location The location dimension specifies the memory region in which the buffer to be overflown is located. Valid options are `stack`, `bss`, `data`, and `heap`. As our approach is not designed for detecting OOB writes on the heap, we only use the first three options.

Target Code Pointer The target code pointer dimension specifies which type of code pointer will be overwritten to divert control flow during the attack. The testbed supports five options, all of which we evaluate on. Particularly the `structfuncptr` option is interesting, as it causes an intra-object OOB write, the detection of which we found not to be supported by many existing sanitizers, as described in Chapter 3.

Overflow Technique The overflow technique dimension specifies whether the attack performs a single overflow that directly overwrites the target code pointer or an indirect overflow that leverages the first overflow to perform a second overflow with which the target code pointer is then overwritten. Since our scope is limited to primary OOB writes, as described in Section 5.1.1, we only evaluate on direct overflows.

Attack Code The attack code dimension specifies the type of payload used by the attack. Options include shellcode, return-to-libc, and return-oriented programming, among others. Since, as outlined in Section 5.1.1, detecting secondary OOB writes is out of scope for us, and different parameters for this dimension will only show their effects after the initial OOB write, we do not vary this parameter during the evaluation.

Abused function The abused function dimension specifies the function by which the OOB write is performed. Possible options are nine different C standard library functions and a custom function `homebrew`, which copies data in a loop similarly to `memcpy`, but without requiring a call to an external library. As it is desirable for an

OOB write detection approach to function correctly regardless of how the write is caused, we evaluate on all options of this parameter.

The restriction to a subset of the parameters and options leaves us with 122 test cases. A tabular overview of the parameters and options that are relevant for us is presented in Table 7.1.

Dimension and param.		Rationale
Location	`stack`	The buffer to be overflown is located on the stack
	`bss`	The buffer to be overflown is located in the `.data` section
	`data`	The buffer to be overflown is located in the `.bss` section
Target Pointer	`ret`	Saved instruction pointer at the bottom of the stack frame
	`baseptr`	Saved base pointer at the bottom of the stack frame
	`funcptr`	Generic function pointer saved in memory
	`longjmp`	Saved instruction pointer in a `jmp_buf` structure, used for `setjmp`/`longjmp` jumping
	`structfuncptr`	Generic function pointer stored in a structure after a vulnerable buffer
Function	`memcpy`	Copies a memory chunk of size n from one buffer to the other
	`strcpy`	Copies a string from one buffer to the other, terminating only on null char
	`strncpy`	Like `strcpy`, but copies at most n bytes of the string
	`sprintf`	Writes a format string to a specified buffer, terminating only on null char
	`snprintf`	Like `sprintf`, but writes at most n bytes of the format string
	`strcat`	Appends a given string to the end of another string, terminating only on null char
	`strncat`	Like `strcat`, but appends at most n bytes of the string
	`sscanf`	Writes formatted input from a given string to the given dest., term. only on null char
	`fscanf`	Like `sscanf`, but reads input from a stream
	`homebrew`	Custom implementation of `memcpy`, uses no library calls

Table 7.1: RIPE parameters varied for testing OOB write detection performance

To distinguish true positives from false positives, we collect the line numbers at which the OOB write is expected to occur from the source code. Then, for each detected memory safety violation, we translate the instruction addresses in the stack trace to line numbers using debug information and check if the expected line number is contained in the stack trace. If so, the detection is considered a true positive. Otherwise, we manually inspect the disassembled binary to identify whether the discovered OOB writes are true or false positives. The motivation for inspecting the entire stack trace instead of just the topmost instruction address is that both ASan and SoftBound perform calls to additional functions for dereference checking, thereby introducing additional entries to the stack trace.

While identifying memory objects affected by a detected OOB write is not a goal of ASan and SoftBound[1], it is an essential feature of our approach. Therefore, we manually verify the correctness of the affected memory objects identified by our approach. As we run around 450 possible test configurations, manually verifying the correctness of identified affected memory objects for each of them is infeasible. Thus, we pick two configurations for each possible combination of location and pointer and manually verify the results using a best-effort approach. To do so, we compare the reported objects with the positioning of objects as reported by the DWARF debug information, as well as the positioning of objects to be expected from the source code and optimization level. In the case of uncertainties about the correctness, we additionally inspect the relevant disassembled code.

As the RIPE testbed exclusively aims to simulate control data attacks, we do not explicitly evaluate our approach's performance for non-control data attacks. However, as our approach does not distinguish between control data and non-control data in the first place, we argue that the results apply to either type of affected data.

7.2.2. Results

The results of our evaluation on the RIPE benchmark for our approach, ASan, and SoftBound are presented in Table 7.2, Table 7.3, and Table 7.4.

Our approach achieves the highest detection rate of all three approaches, detecting the OOB write in 89% of the 452 testable configurations, as shown in Table 7.2. The pattern of failing test cases is clearly visible to be limited to those in which intra-object OOB writes within a structure occur. All such test cases succeed without optimizations, indicating that our approach successfully identified the bounds-narrowing instruction in the compiled code and

[1] In fact, ASan does give an overview of affected objects on the stack. However, as it heavily modifies the memory layout, as described in Chapter 4, this overview is neither usable for comparison to our approach's performance nor for assessing detailed capabilities of a vulnerability.

bounds-narrowing was successfully performed during dynamic analysis. However, with optimizations, this fails in 5 out of 9 cases, thereby preventing detection. A closer investigation of the failing test cases reveals that, in each case, the compiler fails to attach the debug information of the corresponding IR instruction to the bounds-narrowing instruction in the compiled code, thereby preventing bounds-narrowing from being performed.

Manually validating our approach's overview of memory objects affected by the vulnerabilities in the test cases marked in Table 7.2 uncovers a varying quality of results. The ranges affected by the OOB write are correctly identified in all cases, just like the affected stack frames. At `-O0`, all memory objects affected by the OOB write are correctly identified and listed, regardless of whether it occurs on the stack, in the `.data`, or the `.bss` section. At `-O1`, `-O3`, and `-Oz`, however, several objects located on the stack are not recognized as source code-level objects and therefore incorrectly identified as unknown space or padding. This concerns three out of twelve stack objects at `-O1` and four out of twelve at `-O3` and `-Oz`.
In the global sections, on the other hand, all objects are correctly identified. Investigating the cause for the missing stack objects reveals that the compiler produces incorrect debug information, describing the position of some objects as solely in registers while they are, in fact, located in memory over some of their lifetime. Although such missing debug information, in principle, has the potential to prevent us from detecting an OOB write in the first place by causing pointee identification to fail, all objects for which we discovered missing location information here are 8-byte automatic variables, which are typically only modified by innocuous independent writes.

When compiled without instrumentation and aggressive optimizations using `-O3`, the compiler substantially modifies the `homebrew` function to decrease its running time. While the function is designed to copy one byte at a time, the optimizations cause it to be split into multiple loops, each of which copies a different number of bytes per iteration. For our test case, two such loops are used for copying. The first loop uses eight `xmmmov` instructions to copy 128 bytes with each iteration, while the second loop uses two `xmmmov` instructions to copy 32 bytes. This substantially reduces the number of instructions to be executed but also causes our analysis results to be more difficult to interpret, which show OOB writes occurring at ten different locations, each location writing multiple non-adjacent 16-byte chunks.

Lastly, it is not surprising that our approach appears to be the only one whose detection performance is affected by compiler optimizations.

AddressSanitizer detects the OOB write in 70% of the 341 testable configurations, as shown in Table 7.3, placing it second in our comparison. However, a larger number of test cases cannot be executed due to ASan's tendency to rearrange the order of objects on the stack, which we described in Chapter 4. This is because the RIPE benchmark generally relies on buffer overflows, thereby requiring the object to be overflown to be placed at a higher address than the object overflowing. When overlooking the results for `sscanf` and `fscanf`, the detection pattern appears relatively clear. All inter-object OOB writes are successfully detected, while intra-object OOB writes remain undetected, as is to be expected by an approach based on the insertion of red zones between memory objects. For `sscanf` and `fscanf`, however, the results are surprising. Here, the OOB writes in the intra-object test cases are detected, while the OOB writes in the test cases that overwrite the saved instruction pointer and a longjmp buffer passed as a function parameter remain undetected. Particularly the former is surprising, as ASan does not insert red zones between fields of structures and, as such, should not be capable of detecting intra-object OOB writes. While identifying the exact cause for this proved to be difficult due to the opaque instrumentation applied by ASan on a machine code level, we believe that these observations are connected to ASan's tendency to rearrange objects on the stack, causing sequential writes to take place in a way not accounted for by the testbed.

SoftBound achieves the lowest detection rate for the RIPE benchmark, detecting the OOB write in 34% of the 452 testable configurations, as shown in Table 7.4. Again, a clear pattern in the detection results is visible. Contrary to what the description of SoftBound's design suggests [41], the publicly available implementation of SoftBound appears to be incapable of detecting intra-object OOB writes, with none of the OOB writes in corresponding test cases being detected. Furthermore, SoftBound's reliance on wrapper functions for bounds-checking writes in external library functions is evident. As the publicly available implementation of SoftBound only comes with wrappers for `memcpy`, `strcpy`, and `strncpy`, OOB writes in other external functions remain entirely undetected.

The test cases in which the stack frame base pointer is overwritten by a function other than `memcpy` or `homebrew` are generally reported as impossible, as all other functions operate on strings and therefore terminate writing upon encountering a null byte. Since the stack frame base pointer is to be overwritten with a valid pointer to divert execution, the 48-bit valid address space combined with 64-bit pointers makes the inclusion of null bytes inevitable.

For any test case using `sprintf`, compiler optimizations replace the call `sprintf(dst, "%s", src)` with the equivalent call `strcpy(dst, src)`. Such test cases can therefore not be evaluated when using optimizations.

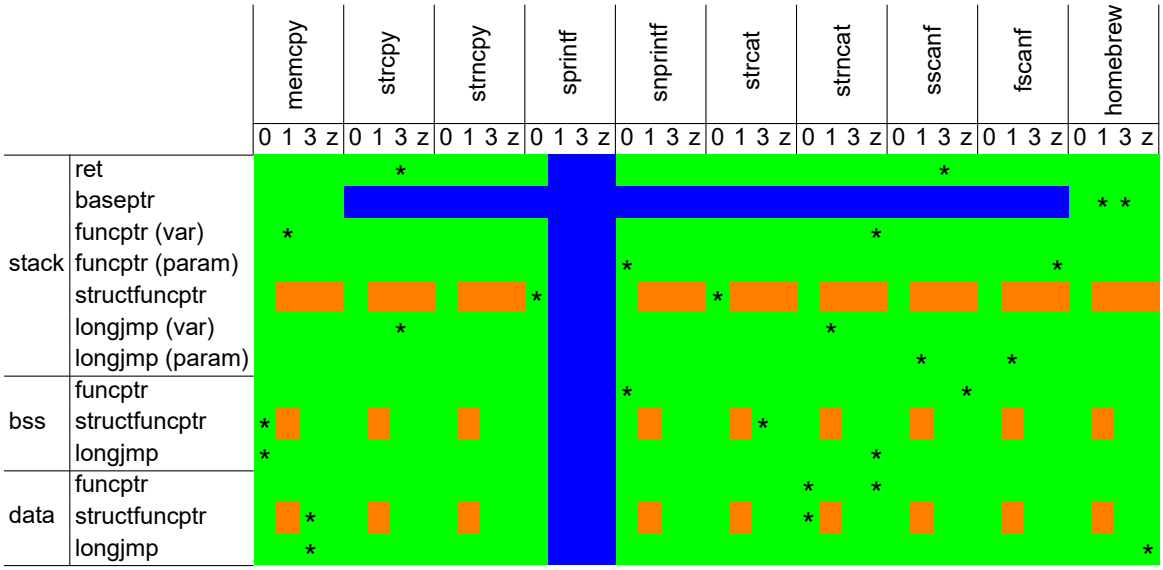


Table 7.2: Detection performance of our approach for dependent OOB writes in RIPE testbed at different optimization levels

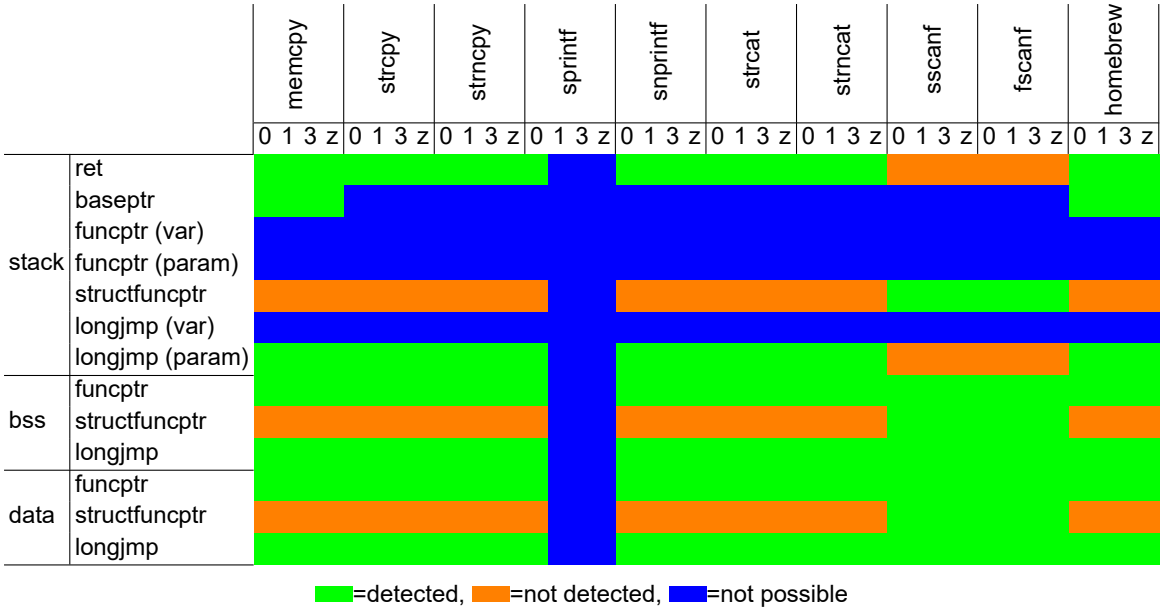


Table 7.3: Detection performance of ASan for dependent OOB writes in RIPE testbed at different optimization levels

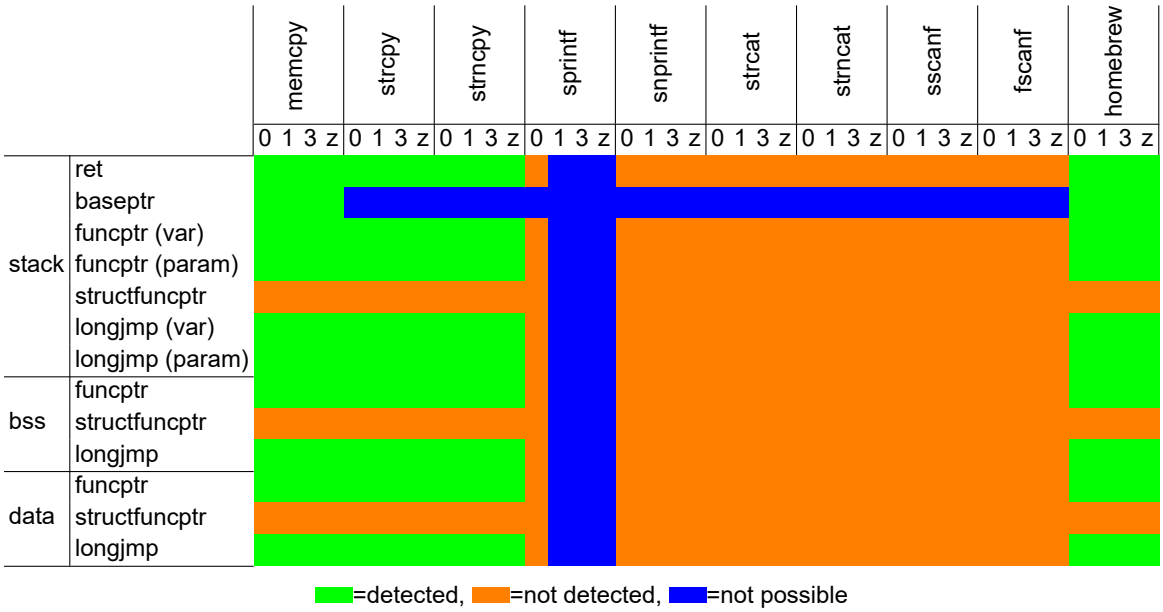


Table 7.4: Detection performance of SoftBound for dependent OOB writes in RIPE testbed at different optimization levels

7.3. Independent Writes

As our approach's procedure for independent OOB write detection is largely disjoint from the procedure for dependent OOB write detection, explicitly measuring its detection performance is essential. Similar to dependent writes, we do so by means of a testbed.

7.3.1. Setup

In classical compile-time instrumentation-based approaches, independent OOB write detection is typically simple due to the knowledge of the intended destination object combined with the capability of invasively modifying the code. Thus, to the best of our knowledge, there are no publicly available testbeds specifically for detecting independent OOB writes. Therefore, we designed our own testbed, which comprises four parameter dimensions and 44 test cases and is made publicly available along with our implementation. The testbed is designed similarly to RIPE, combining all test cases in a program compiled to a single binary and accepting command line arguments to specify which test case is to be executed.
Contrary to RIPE, our testbed does not attempt to perform full exploitation and spawn a shell but terminates after performing the OOB write that would be necessary for this. As we defined our scope in Section 5.1.1 to be limited to primary OOB writes, this behavior is sufficient for testing our approach.

The **overwrite type** dimension specifies the type of OOB write to be performed. Valid options are `continuous`, which causes a simple continuous buffer overflow that overwrites everything between the start of the buffer and the target code pointer, and `jumping`, which only performs a single write, using an offset from the buffer that causes the first byte of the target code pointer to be overwritten, not affecting any other data.

The **location** dimension specifies the memory region in which the buffer used for the OOB write is located. Valid options are `stack`, `bss`, and `data`.

The **target code pointer** dimension specifies which type of code pointer will be overwritten during the OOB write. Although we could also overwrite any non-control data, as we do not aim for actual exploitation of the program, we opted to stay aligned with the RIPE testbed and support the same options.

The **context** dimension specifies the context in which the OOB write is performed. The `iso` option causes the OOB write to occur in a function that has the sole purpose of causing an OOB write and does not contain any additional logic. For the `svd` option, on the other hand, the code in which the OOB write occurs is integrated into unrelated logic containing many non-innocuous independent writes. For this, we chose the implementation of an algorithm for Singular Value Decomposition of matrices. The motivation for this second option is to allow testing in a scenario where a larger number of independent writes are present in the same function, potentially making it more difficult for our approach to match ASM writes correctly to corresponding IR writes.

Similar to the dependent write evaluation using RIPE, we distinguish true positives from false positives by inspecting the stack trace at detection time. Furthermore, we again validate the correctness of identified memory objects by selecting one configuration at random for each combination of location, target pointer, and context and manually validating the results.

7.3.2. Results

The results of evaluating our approach, ASan, and SoftBound on our testbed are presented in Table 7.5.

Our approach successfully detects the OOB write in 95% of the test cases. In 13% of the configurations, however, a false positive is detected. Detection succeeds for all inter-object OOB writes and only fails for some intra-object OOB writes on the stack and in the `.data` section.
Investigating the cause for this reveals the responsible fault to occur during the IR analysis, where tracing the write's declaration chain terminates prematurely. As shown in Figure 7.1, the source pointer operator of the bounds-narrowing GEP instruction is cast from the original structure, consisting of an array of 255 bytes and a pointer, to an array of 256 bytes. This causes our analysis pass to conclude that this is not a relevant bounds-narrowing instruction, as the subject is not an object we consider composite. This is a typical example of the large variety of peculiarities that can occur in optimized code and should ideally be accounted for by our approach.

```
@data_struct = dso_local global { <{ i8, [255 x i8] }>, i32 (i8*)* }, align 8

%arrayidx370 = getelementptr inbounds [256 x i8],
                   [256 x i8]* bitcast ({ <{ i8, [255 x i8] }>, i32 (i8*)* }* @data_struct to [256 x i8]*),
                   i64 0, i64 %idxprom369
%store i8 71, i8* %arrayidx370, align 1
```

Figure 7.1: Static write and bounds-narrowing instruction that cause a false negative for our approach in our dependent write testbed

Investigating the false positive detected by our approach for configurations with the SVD context at optimization level `-O1` reveals the machine code generated from a part of the SVD algorithm to be responsible, which is

executed at every test case using the SVD context. In the responsible code, displayed in Figure 7.2, our implicit assumption that every pointer points to its intended pointee object upon creation is violated. In the `lea` instruction at `0x402ece`, a pointer is created that is supposed to point to a buffer on the stack. However, it does not do so until the next instruction at `0x402ed2` adds an offset to it, causing our approach to taint the pointer with an incorrect object. The consequence is a false positive detection in the `memset` call at `0x402ef5`, where the pointer is dereferenced for writing. It is unclear why the compiler split the pointer creation into two instructions instead of using `lea rdi, [rax+rbp-0x1a0]` as we expected, which would require fewer bytes to represent and presumably execute faster.

```
402ECE  lea     rdi, [rax+rbp]
402ED2  add     rdi, -1A0h      ; s
402ED9  mov     esi, r8d        ; c
402EDC  mov     rbx, r9
402EDF  mov     [rbp-188h], r10
402EE6  mov     [rbp-180h], r11
402EED  movsd   qword ptr [rbp-130h], xmm1
402EF5  call    _memset
```

Figure 7.2: Assembly snippet causing a false positive in our independent write testbed

Manually validating the overview of affected objects generated by our approach yields similar results as the previous experiment. While the results appear correct and complete for `-O0`, several objects are missing from the summary at `-O1`, `-O3`, and `-Oz` for OOB writes affecting the stack. A closer investigation again reveals incomplete debug information generated by the compiler to be at fault. Again, this affects only variables of at most 8 bytes, thereby limiting the consequences to incomplete reporting.

As in the previous experiment, our approach is the only one affected by compiler optimizations.

AddressSanitizer detects the OOB write in 36% of the 176 test case configurations, placing it last in the comparison. This poor result is mainly caused by ASan's reliance on red zones, as outlined in Chapter 2, which prevents it from detecting OOB writes that jump over the red zones and only affect addresses occupied by other objects. Furthermore, the results again show that ASan cannot detect intra-object OOB writes.

SoftBound successfully detects the OOB write in 73% of the test cases. All inter-object OOB writes are successfully detected, while all intra-object OOB writes remain undetected.

7.4. Real-world Programs

To show that our approach does not cause an unacceptably high number of false positives in larger programs and still manages to reliably detect OOB writes in them, we evaluate it on several real-world vulnerabilities found in open source software. Additionally, we run each program under test with a benign input to assess our approach's tendency to report false positives.

7.4.1. Setup

We selected real-world vulnerabilities for evaluation based on the following criteria.

- The program is open source.
- The component in which the vulnerability occurs is written in the C programming language.
- The vulnerability is an out-of-bounds write in the `bss` or `data` section or on the stack.
- A proof-of-concept input that triggers the vulnerability is publicly available or can easily be constructed from the vulnerability's description.

These selection criteria led us to evaluate our approach on the three real-world vulnerabilities described below.

As we found two of the three vulnerabilities to cause very early program termination, thereby resulting in the execution of only a small part of the program, we additionally run each program with a benign input to assess our approach's tendency to report false positives.

Statistics

As our approach comprises multiple sub-challenges, the solution for some of which is potentially error-prone, we show the performance of our approach's internal components by collecting several additional statistics.

To show the performance of our static analysis implementation on each program under test, we present statistics on the performance of non-innocuous write matching and bounds-narrowing instruction matching. Statistics on the former provide insights on how many independent writes in the program our approach is able to bounds-check

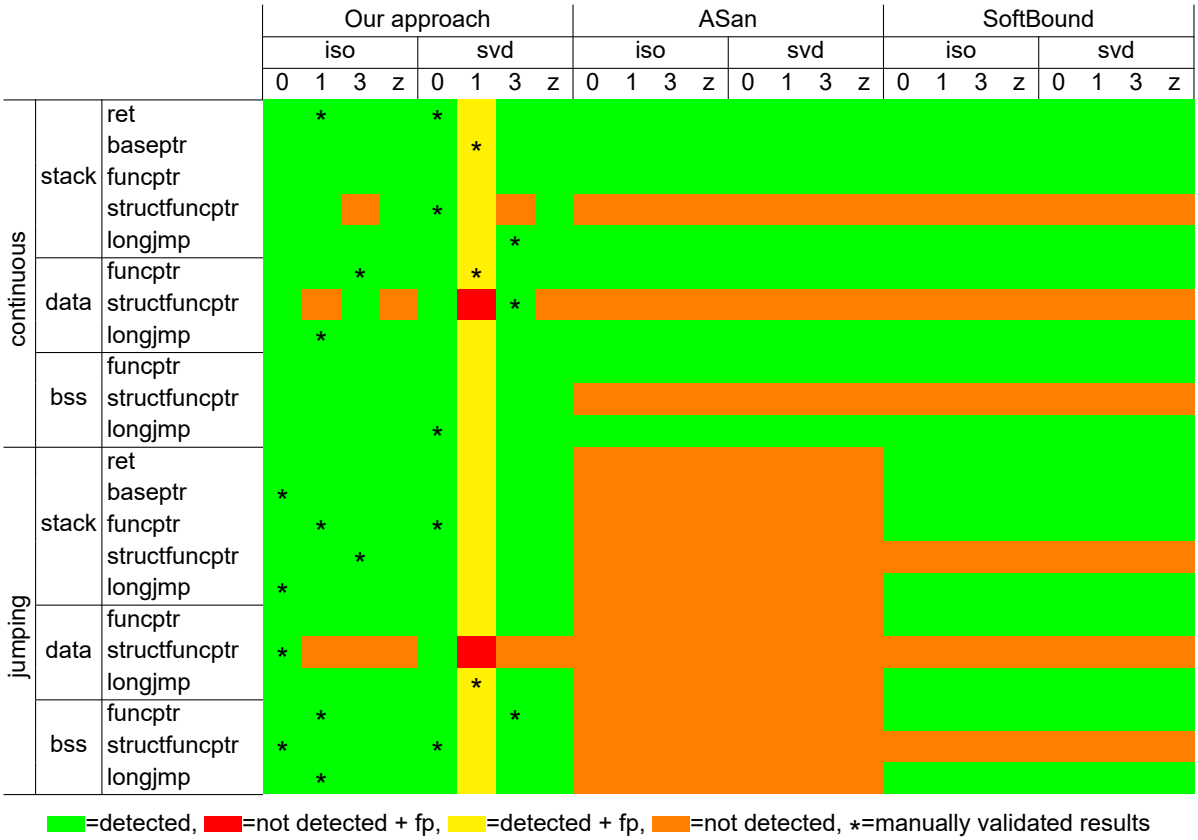


Table 7.5: Detection performance of our approach, ASan, and SoftBound for independent OOB writes in our testbed

during dynamic analysis and how compiler optimizations influence this ability. Similarly, statistics about the bounds-narrowing instruction matching performance allow us to assess our approach's ability to monitor for intra-object OOB writes.

Furthermore, we show the quality of the extracted memory layout by presenting the mean share of uncharted space within the stack frames of each program's functions at all four optimization levels. To compute this, we determine the size of each function's stack frame from the disassembled code and prune off space used for the saved `rbp` or `rip`, as well as any space used for storing caller-saved registers. If any space is left after this, we compute the share of space within the remaining stack frame that is never occupied by any automatic variable we extracted from the DWARF debug information. For this, we only consider uncharted chunks of at least 8 bytes, as we found smaller chunks to serve as padding in most cases.
Furthermore, we do not consider a chunk if it is the topmost chunk of the stack frame[2], which is highly likely to be used for passing function arguments on the stack. As the DWARF debug information for such arguments is attached to the callee, we cannot check this without introducing substantial complexity. Consequently, as this simplification will inevitably cause some uncharted chunks to remain unnoticed, the share of uncharted chunks must be considered a lower bound.

For our dynamic analysis, we collect and present statistics of three categories.
First, we present the number of unchecked non-innocuous, checked independent, and checked dependent writes. Here, the ratio of checked to unchecked writes provides insights into the overall effectiveness of our bounds-checking approach, while the ratio of independent to dependent checked writes allows us to understand the types of memory accesses performed by the program. It is important to note that the number of unchecked writes does not include innocuous independent writes, which are deliberately excluded from checking due to their inability to cause OOB writes in our scope. Instead, it describes the number of writes missed entirely by our approach, e.g., because an independent write was not successfully matched during static analysis or a pointer was not tainted.
Secondly, we present statistics on the success ratio of bounds narrowing. We decompose the number of encountered corresponding instructions into those at which bounds-narrowing is deemed unnecessary because the pointer does not reference an address in a section monitored by us (stack, `.data`, `.bss`), those at which bounds-narrowing failed, and those at which it succeeded. The latter is again decomposed into internal and emitting bounds-narrowing

[2] As the stack grows towards lower memory addresses, the topmost chunk is the chunk whose lower bound is equal to `rsp`.

instructions. These numbers provide insights into the frequency with which bounds-narrowing occurs in the program and how reliable our approach for bounds-narrowing performs.

Lastly, we provide insights on the quality of the extracted memory layout by presenting statistics about the successfulness of determining pointees from the recorded memory layout, which we observed to occasionally fail in one of two ways. First, if no object is located at the address pointed to, this indicates incomplete information provided about the memory layout by the DWARF debugging information, potentially due to objects created by the compiler that do not correspond to any source code-level variables. Second, if determining the pointee fails because multiple objects are recorded as being located at the address, this is caused by our simplifying assumption of the lifetime of objects being describable by a single interval. Thus, it serves as an indicator of this decision's severity.

7.4.2. libxml (CVE-2017-9047)

This vulnerability in the *libxml* XML document parsing library is a stack-based buffer overflow located in the logic for checking whether a given XML document is well-formed with regard to the accompanying Document Type Definition (DTD). It was discovered in 2017 through directed fuzzing with AFLGo [59] and patched six weeks later.

The vulnerable code of the function `xmlSnprintfElementContent` is presented in Figure 7.3. This code is intended to append two strings, contained in `content->prefix` and `content->name`, to the string located in the char array `buf`, a pointer to which is passed as an argument to the function. From line 6 to 13, appending the `prefix` field to the buffer takes place without causing any issue. Prior to appending the `name` field to the buffer between lines 14 and 19, however, the size check in line 14 does not consider the fact that the length of the string in `buf` is not `len` anymore after appending `prefix` to it. Instead, the previous length is reused, potentially causing the call to `strcat` in line 19 to overflow `buf` by `strlen(content->prefix)+1` bytes. Afterwards, a second OOB write of one byte can occur in line 24.

Successfully detecting this OOB writes requires our approach to perform several tasks. First, it must correctly determine `buf` as the pointee object when its pointer is created in the function that calls `xmlSnprintfElementContent`. Then, the bounds of the object must be attached to the pointer by tainting it. As `strcat` is a standard library function intercepted by us, as described in Section 6.4.6, our approach must then detect the call to `strcpy` and obtain the source and destination arguments from the corresponding registers. From the positions of the first terminating null character in the strings pointed to by the source and destination pointers, the prospective memory addresses to be overwritten must be computed. Afterwards, the taint of the destination pointer must be correctly resolved to the bounds of the `buf` object, allowing for checking whether any written byte will be out-of-bounds.

To assess our approach's behavior during normal execution of xmllint, we run it on a benign 340-byte XML file.

```
1  void xmlSnprintfElementContent(char *buf, int size, xmlElementContentPtr content, int englob) {
2    int len = strlen(buf);
3    <omitted>
4    switch (content->type) {
5      case XML_ELEMENT_CONTENT_ELEMENT:
6        if (content->prefix != NULL) {
7          if (size - len < xmlStrlen(content->prefix) + 10) {
8            strcat(buf, " ...");
9            return;
10         }
11         strcat(buf, (char *) content->prefix);
12         strcat(buf, ":");
13       }
14       if (size - len < xmlStrlen(content->name) + 10) {
15         strcat(buf, " ...");
16         return;
17       }
18       if (content->name != NULL)
19         strcat(buf, (char *) content->name);
20       break;
21     <omitted>
22   }
23   if (englob)
24     strcat(buf, ")");
25   <omitted>
26 }
```

Figure 7.3: Vulnerable code causing CVE-2017-9047 in libxml

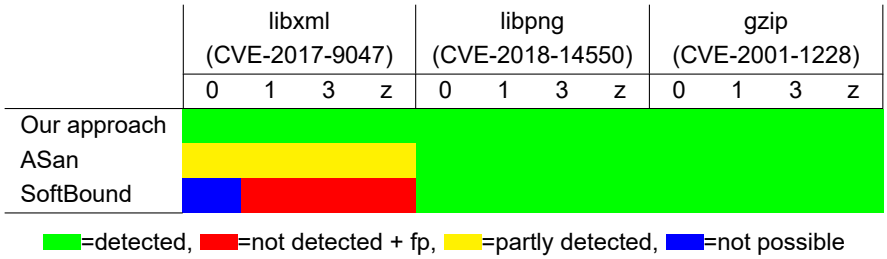


Table 7.6: Detection performance of our approach, ASan, and SoftBound for each of the three real-world vulnerabilities

		gzip				pnm2png				xmllint			
		0	1	3	z	0	1	3	z	0	1	3	z
Non-innocuous write	Checked (dependent)	2	1	1	1	31	31	0	31	5824	5825	5935	5885
	Checked (independent)	0	0	0	0	0	0	30	0	71	71	1	71
	Unchecked	0	0	0	0	0	0	0	0	0	0	152	162
	Total	2	1	1	1	31	31	30	31	5895	5896	6088	6118
Bounds narrowing	Failed - Type mismatch	0	0	0	0	0	0	0	0	15	10	0	7
	Skipped - Unnecessary	0	0	0	0	0	0	0	0	136	5777	5746	5823
	Successful (internal)	0	0	0	0	0	0	0	0	0	0	7	7
	Successful (emitting)	0	0	0	0	0	0	0	0	0	0	0	0
	Total	0	0	0	0	0	0	0	0	151	5787	5753	5837
Pointee inference from memory	Failed - No known object at address	0	2	4	4	0	0	0	0	0	1	22	29
	Failed - Multiple objects at address	0	0	0	0	0	0	0	0	0	0	8	0
	Successful	18	15	15	16	5	5	4	5	6080	360	504	508
	Total	18	17	19	20	5	5	4	5	6080	361	534	537

Table 7.7: Statistics collected during the dynamic analysis of the real-world vulnerabilities using a PoC input

Results

As shown in Table 7.6, our approach successfully detects both OOB writes caused by the vulnerability at all four optimization levels and does not yield any false positives for the PoC or the benign input.

As presented in Figure 7.5, independent write matching during static analysis achieves a 100% matching rate without optimizations. However, the matching rate decreases with higher optimization levels, achieving only 75% at `-O3`. This is caused by overlapping objects in our recorded memory layout (0-17%) and a lack of line number debug information generated by the compiler (5-9%).

Bounds-narrowing instruction matching, shown in Figure 7.6, also performs best without optimizations but generally has a substantially lower matching rate. Here, the most frequent reason for failed matching is a lack of line number debug information generated by the compiler. However, with enabled optimizations, our simplification of omitting dynamic bounds-narrowing instructions is responsible for up to 10% of failed matches.

As shown in Table 7.7 and Table 7.8, the majority of checked writes is dependent. The share of checked writes ranges from 100% down to 31% when executing a benign input at `-O3`. The latter appears to be caused by one or more independent writes not being matched during static analysis. Most bounds-narrowing instructions encountered during execution concern pointers to unmonitored sections, presumably mainly the heap. While several internal bounds-narrowing instructions are encountered, no emitting ones are executed. Lastly, determining a pointer's intended pointee object from the extracted memory layout is fairly successful, ranging from a 100% success rate at `-O0` to a 95% success rate at `-O3` for the PoC input. Most failures can be attributed to a lack of available information about the object at this location, caused by incomplete debug information due to optimizations. Our simplifying assumption of memory object's lifetimes being representable by a single interval is responsible for at most 1% of failed pointee inferences.

		gzip				pnm2png				xmllint			
		0	1	3	z	0	1	3	z	0	1	3	z
Non-innocuous write	Checked (dependent)	297	353	351	293	88	58	27	83	186	173	169	167
	Checked (independent)	8350	6651	6401	6827	0	0	11	0	77	77	0	77
	Unchecked	813	1390	813	573	0	6	0	7	0	0	77	0
	Total	9460	8394	7565	7693	88	64	38	90	263	250	246	244
Bounds narrowing	Failed - Type mismatch	0	0	0	0	0	0	0	0	15	10	0	7
	Skipped - Unnecessary	0	0	0	0	208	207	207	208	234	139	86	157
	Successful (internal)	3	3	0	0	9	6	6	6	0	0	0	0
	Successful (emitting)	5	3	1	3	1	1	1	1	0	0	0	0
	Total	8	6	1	3	218	214	214	215	249	149	86	164
Pointee inference from memory	Failed - No known object at address	0	2	4	4	0	1	1	1	0	1	1	9
	Failed - Multiple objects at address	0	0	0	0	0	0	1	0	0	0	8	0
	Successful	1971	644	640	584	450	431	429	430	865	704	647	661
	Total	1971	646	644	588	450	432	431	431	865	705	656	670

Table 7.8: Statistics collected during the dynamic analysis of the real-world vulnerabilities using a benign input

The mean share of uncharted stack frame space is presented in Figure 7.4. While the large share of uncharted stack frame space at `-O0` is surprising at first, a closer investigation reveals the cause for this to be the compiler's tendency to write intermediate results to memory instead of storing them in registers. This causes the creation of a large number of objects that do not have a corresponding source code-level object and, therefore, no representation in DWARF. With an increasing optimization level, the share of uncharted stack space decreases, reaching just 4% at `-O3`.

We manually validated the correctness of the identified affected objects by inspecting the disassembled code. The truncated results of our approach when executing the PoC input with `-O1` optimizations are presented in Figure A.1. In total, the OOB write fully overwrites the stack frames of six functions and partially overwrites those of two functions, affecting a multitude of source code-level objects, as well as the saved instruction pointer in many cases. As in the first two experiments, we identified a number of smaller objects whose location in memory is not properly recorded in the DWARF debug information. However, most space marked as *unknown/padding* is used for storing the values of caller-saved registers or as padding.

While the OOB write affects many objects and stack frames at `-O0` and `-O1`, compiling with `-O3` or `-Oz` causes it to only affect a single 5000-byte buffer, as shown in Figure 6.3. The cause is a rearrangement of stack objects at this optimization level. While the 5000-byte buffer is placed before the overflowing buffer at `-O0` and `-O1`, thus leaving it unaffected, it is placed right after the overflowing buffer at `-O3` and `-Oz`. This is a good example of the stringent necessity to evaluate the effects of an OOB write by inspecting the program using the same binary used in practice. While the vulnerability can easily be used to divert control flow at `-O0` and `-O1` if security measures like canaries are disabled, this is likely impossible at `-O3` and `-Oz`, assuming the PoC cannot be modified to overwrite a larger range.

ASan successfully detects the first OOB write in the vulnerable function at each optimization level we consider. The second OOB write, however, is not detected. The cause is likely that the destination address of the second OOB write is a memory location that is not used as a red zone by ASan but is part of another object in memory. As such, the second OOB write effectively jumps over the inserted red zone, making it undetectable by ASan.

When attempting to instrument libxml with SoftBound using no optimizations (`-O0`), the build process fails with the SoftBound instrumentation pass throwing an exception. Thus, we cannot test SoftBound's detection performance at this optimization level. At `-O1`, `-O3` and `-Oz`, compilation is successful but SoftBound's instrumentation appears to break the program at run time. Running the PoC input causes SoftBound to report an OOB write early in the program before the vulnerable location and terminates with a segmentation fault immediately afterwards. Investigating this in more detail reveals that the reported OOB write is caused by SoftBound's pointer dereference checking function being called with a null pointer for both the base and the bound.

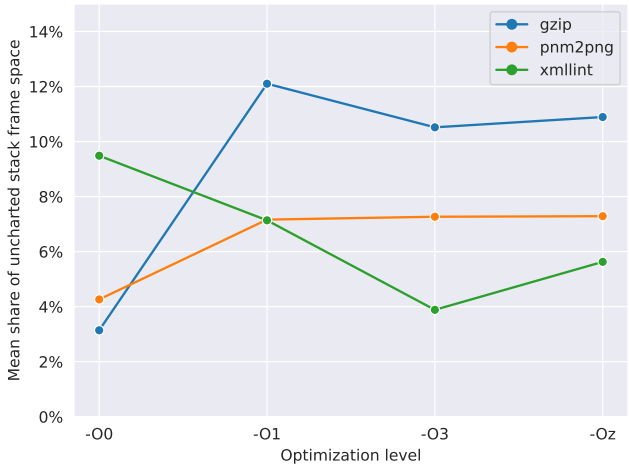


Figure 7.4: Mean uncharted stack frame space for each program and optimization level

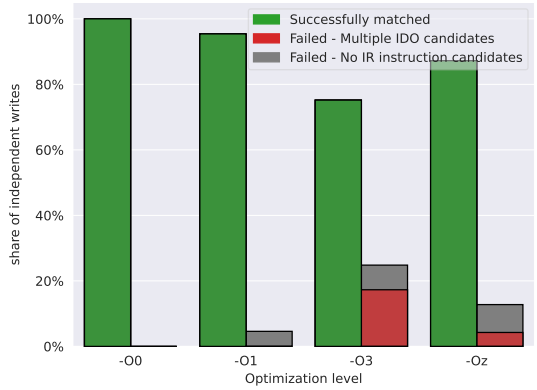


Figure 7.5: Independent write matching performance for xmllint

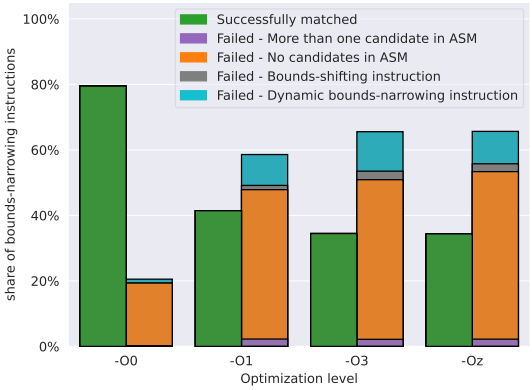


Figure 7.6: BNI matching performance for xmllint

7.4.3. libpng (CVE-2018-14550)

This stack-based buffer overflow is located in the *pnm2png* tool, which is part of the `libpng` image handling library, and serves to convert images from the PNM to the PNG format. It was discovered in July 2018 and fixed in April 2019 [60].

The vulnerable code of the function `get_token` is presented in Figure 7.7. It is intended to read from the file stream created by opening the PNM file and write the next token - a byte sequence ending with a newline or space - into the buffer specified by a pointer passed to the function. While this buffer always has a fixed size of 16 bytes, the token is assumed at most 15 bytes large but never checked for this. Thus, a token larger than 15 bytes will cause the buffer to overflow at line 14, while a token larger than 16 bytes will also cause it to overflow in the loop at line 11. Although the cause for the vulnerability is slightly different from the vulnerability in libxml, the demands on our approach for detecting the OOB write are similar. The main difference is that the OOB write is not performed by a standard library function but by a loop in the vulnerable function. Thus, our approach must monitor for writes to memory and resolve the bounds from the pointer's taint upon a write to it in line 11 in every iteration.

To assess our approach's behavior during normal execution of pnm2png, we use it to convert an 81kB PNM image to PNG and write the result to disk.

```
1  void get_token(FILE *pnm_file, char *token) {
2    int i = 0;
3    int ret;
4    <omitted>
5    /* read string */
6    do
7    {
8      ret = fgetc(pnm_file);
9      if (ret == EOF) break;
10     i++;
11     token[i] = (unsigned char) ret;
12   }
13   while ((token[i] != '\n') && (token[i] != '\r') && (token[i] != ' '));
14   token[i] = '\0';
15   return;
16 }
```

Figure 7.7: Vulnerable code causing CVE-2018-14550 in libpng

Results

As shown in Table 7.6, our approach successfully detects both OOB writes in pnm2png without raising any false positives at any tested optimization level. For the benign input, however, false positives occur at optimization levels `-O1`, `-O3`, and `-Oz`. We analyze the cause for this below.

During the static analysis, our approach successfully matches most independent writes to their counterparts in the LLVM IR, as shown in Figure 7.8. Although the matching rate decreases with higher optimization levels, it never falls below 90%. On the other hand, the matching performance of the bounds-narrowing instructions, shown in Figure 7.9, is substantially lower. Reaching from 64% at `-O0` down to 30% at `-O3` and `-Oz`, the most frequent reason for failed matching is again a lack of line number debug information in the binary, followed by the disregard for dynamic BNIs, which make up as much as 20% of BNIs in the optimized binary.

During dynamic analysis, as presented in Table 7.8 and Table 7.7, most checked writes are dependent. Only at `-O3` a substantial number of independent writes is checked, caused by inlined functions. The share of checked writes never falls below 90%. While several bounds-narrowing instructions were encountered when using a benign input, most of which were skipped due to a pointer to an unmonitored section, the PoC input does not cause any bounds-narrowing instructions to be executed at all. Together with the fact that the number of times a pointee inference from the memory layout was attempted is also substantially lower than for the benign input, this is a strong indicator of the vulnerability's shallowness.

The share of uncharted stack space, presented in Figure 7.4, amounts to only around 4% at `-O0` and increases to roughly 7% with optimizations. This increase is presumably caused by the introduction of objects that have no corresponding source code-level object due to optimizations and by wrongfully omitted location descriptions, one of which we observed in our approach's report on the vulnerability capabilities.

Comparing the memory objects reported as affected by our approach with the memory layout that can be inferred from the disassembled binary reveals that the results are correct at all optimization levels, despite again missing an 8-byte object not present in DWARF when employing optimizations. While the 14-byte overflow of the `type_token` buffer caused by the PoC input only affects a single byte of a source code-level object at `-O0`, it entirely overwrites three objects at `-O1`. However, as the PoC input can presumably be adjusted to overwrite an arbitrary number of bytes, this vulnerability can potentially be used to divert control flow.

ASan successfully detects OOB writes at both locations for all tested optimization levels. For the OOB write occurring in the loop, however, only the first one-byte OOB write is reported. Further OOB writes at the same location in the loop are omitted. However, this is presumably because ASan deliberately omits subsequent writes at the same location to keep the results concise.

SoftBound, on the other hand, successfully detects every single OOB write at each optimization level, both within the loop and outside of it.

Running an optimized version of pnm2png with a benign input using our approach causes non-existent OOB writes to be reported in the function `png_create_png_struct`. The code responsible for this is presented in Figure 7.10. Figure 7.11 shows the corresponding assembly code, as generated at optimization levels `-O1` and `-O3`. In the source code, three 64-bit fields of `create_struct` are zero-initialized. This zero-initialization is optimized by the compiler to occur by creating a pointer to the first field of the triplet `longjmp_fn` at `0x408576`. This pointer is then used at `0x4085A3` to zero-initialize both the `longjmp_fn` and the adjacent `jmp_buf_ptr` field using a 128-bit SSE `movups` instruction - causing the first false positive. In the next instruction, the same pointer is then used to zero the third field `jmp_buf_ptr`, causing the second false positive. When compiled with `-Oz`, the pointer is also used for initializing the three fields of `zstream`, causing two further false positives.

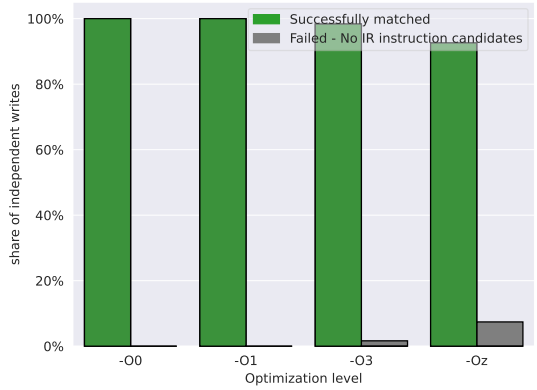


Figure 7.8: Independent write matching performance for pnm2png

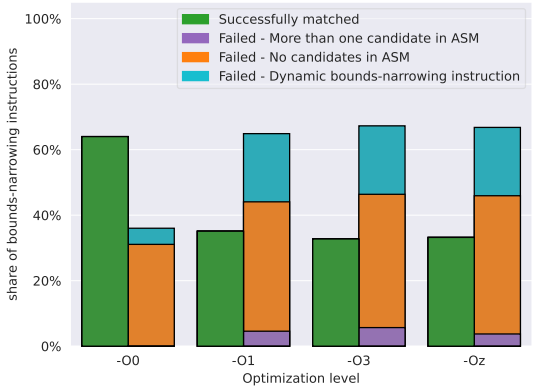


Figure 7.9: BNI matching performance for pnm2png

```
create_struct.zstream.zalloc = png_zalloc;
create_struct.zstream.zfree = png_zfree;
create_struct.zstream.opaque = png_ptr;

#ifdef PNG_SETJMP_SUPPORTED
/* Eliminate the local error handling: */
create_struct.jmp_buf_ptr = NULL;
create_struct.jmp_buf_size = 0;
create_struct.longjmp_fn = 0;
#endif

*png_ptr = create_struct;
```

Figure 7.10: Code snippet of libpng causing false positives

```
408560   lea     rdi, [rbp+create_struct] ; png_ptr
408567   mov     esi, 4E8h       ; size
40856C   call    png_malloc_warn
408571                           ; png_structrp
408571   test    png_ptr, png_ptr
408574   jz      short loc_40851B
408576   lea     rcx, [rbp+create_struct.longjmp_fn]
40857D   mov     rdi, png_ptr    ; dest
408580                           ; png_structrp
408580   mov     rbx, rax
408583   mov     [rbp+create_struct.zstream.zalloc], offset png_zalloc
40858E   mov     [rbp+create_struct.zstream.zfree], offset png_zfree
408599   mov     [rbp+create_struct.zstream.opaque], rax
4085A0   xorps   xmm0, xmm0
4085A3   movups  xmmword ptr [rcx], xmm0
4085A6   mov     qword ptr [rcx+10h], 0
4085AE   lea     rsi, [rbp+create_struct] ; src
4085B5   mov     edx, 4E8h       ; n
4085BA                           ; png_structrp
4085BA   call    _memcpy
```

Figure 7.11: Compiled version of the code snippet (`-O1`)

7.4.4. gzip (CVE-2001-1228)

This `.bss` buffer overflow in a rather old version of the gzip compression utility is one of the few global buffer overflows in open source C programs with a publicly available PoC input.
The vulnerable code is presented in Figure 7.12. The gzip executable is called, specifying the paths of one or more files to (de)compress as command line arguments and handles them one-by-one, as shown in lines 8 to 10. To check if a file really exists, `get_istat` is called with a pointer to the char array containing the filename. Here, `strcpy` is invoked to copy the filename to the global char array `ifname`, which has a fixed size of 1024 bytes. As the filename can be arbitrarily long and no size checks are performed, a filename longer than 1023 bytes will cause `ifname` to overflow.
Although the overflowing buffer is a global variable, the usage of `strcpy` to write to it causes a pointer to be created, making this a dependent OOB write that our approach must detect in a fashion similar to the previous two vulnerabilities. The only difference is that our approach must determine the intended destination object of a pointer to the `.bss` section instead of the stack.

To assess our approach's behavior during normal execution of gzip, we instruct it to compress a single 1.2kB text file and write the output to stdout.

Results

As shown in Table 7.6, our approach successfully detects the OOB write caused by the vulnerability at all four optimization levels and does not yield any false positives for the PoC input. For the benign input, however, multiple false positives are reported. We investigate these in detail below.

As presented in Figure 7.13, independent write matching performs well, with less than 5% of independent writes remaining unmatched, even at high optimization levels. On the other hand, bounds-narrowing instruction matching, presented in Figure 7.14, performs exceptionally poorly. Only 10% of BNIs are matched at `-O0` and less than 1% are matched at `-O3`. This is primarily caused by the frequent occurrence of dynamic BNIs, which make up between 70% and 85% of all BNIs. Furthermore, a non-negligible number of bounds-shifting instructions, a potential source of false positives, is found at `-O3`.

```
1  char ifname[1024];
2
3  int main (int argc, char **argv) {
4    int file_count = argc - optind;
5    <omitted>
6    if (file_count != 0) {
7      <omitted>
8      while (optind < argc) {
9        treat_file(argv[optind++]);
10     }
11   }
12   <omitted>
13 }
14
15 local void treat_file(char *iname) {
16     <omitted>
17     /* Check if the input file is present, set ifname and istat: */
18     if (get_istat(iname, &istat) != OK) return;
19   <omitted>
20 }
21
22 int get_istat(char *iname, struct stat *sbuf) {
23   <omitted>
24     strcpy(ifname, iname);
25   <omitted>
26 }
```

Figure 7.12: Vulnerable code causing CVE-2001-1228 in gzip

When processing the PoC input, a remarkably small number of non-innocuous writes is encountered, as evident from Table 7.7. This, together with the fact that not a single bounds-narrowing instruction is encountered and few pointers are attempted to be resolved to an object by using the memory layout, is ample evidence for the shallowness of the vulnerability, which aligns with our manual investigation in Section 7.4.1.

For the benign input, on the other hand, many non-innocuous writes, particularly dependent ones, are encountered and checked. However, up to 16% of writes also remain unchecked. A closer investigation reveals the cause for this to be missed pointer tainting at two locations, which could be mitigated by implementing deferred tainting at function calls, as described in Section 5.7. Table 7.8 shows that running a benign input with gzip results in the largest number of emitting bounds-narrowing instructions we observed in any of the three programs. However, as our handling of 8 out of 12 encountered emitting BNIs causes a false positive afterwards, the benefit of handling emitting BNIs here is questionable.

Manual validation of our approach's output shows that all affected objects are correctly identified at all optimization levels. This again demonstrates that the issues of an incomplete memory layout extraction we previously discovered are limited to the stack.

Although the extraction of stack frame memory layout is irrelevant for detecting and analyzing this vulnerability, we again present the mean shares of uncharted space in Figure 7.4. For this program, the share of uncharted stack space amounts to only around 3% at `-O0` but increases to 12% at `-O1` before slightly decreasing again for higher optimization levels. A brief investigation of the cause of this jump suggests that the optimization of two functions performing extensive data transformation is responsible for this by resulting in a large number of objects that do not correspond to any source code-level objects.

Both ASan and SoftBound successfully detect the vulnerability without raising any false positives.

When running gzip with a benign input using our approach, three false positives are reported at `-O0` and `-O1`, and one false positive is reported at `-O3` and `-Oz`. Investigating these false positives reveals that all of them are caused by our disregard for bounds-shifting instructions. Both functions in which the false positives are reported - `scan_tree` and `gen_codes` - take a pointer to a structure as their first argument. This pointer, which points to a structure located in an array of structures, is then used to access other elements of that array, causing our approach to incorrectly report OOB writes. Thus, these false positives can be prevented by considering arrays of structures as unitary objects. The reason for the absence of two false positives at `-O3` and `-Oz` was found to be aggressive function inlining, which eliminates the `scan_tree` function.

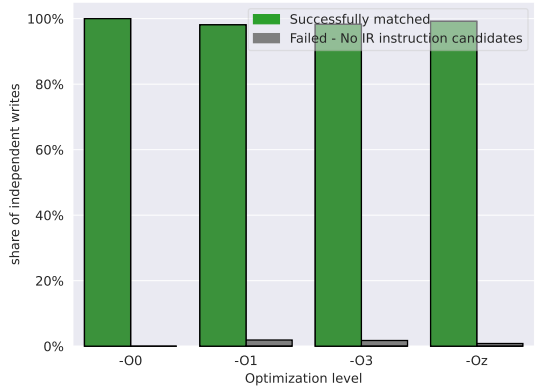


Figure 7.13: Independent write matching performance for gzip

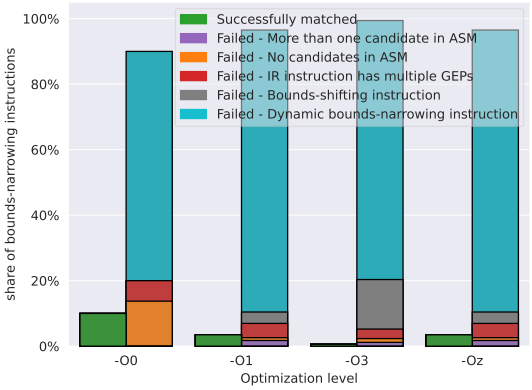


Figure 7.14: BNI matching performance for gzip

7.5. Performance Overhead

Although a low performance overhead was not a goal in our design, and even less so in our implementation, we nevertheless perform experiments to put our approach's performance into perspective and compare it with the overhead incurred by AddressSanitizer and SoftBound.

7.5.1. Setup

We evaluate our approach's performance overhead on each of the three vulnerable real-world programs presented in Section 7.4. As there is no need for spatial memory violations to occur during performance measurements, we provide the programs with the same benign inputs used for assessing the occurrence of false positives in the previous experiment.

We run each of the three programs with ASan, SoftBound, our approach, and natively without any detection approach under each of the four optimization levels described in Section 7.1.3. We run each configuration ten times to combat outliers and consider the mean. Note that we only measure the time required for executing the program, not for compilation or static analysis. All measurements are performed under Ubuntu 20.04 on an Intel Xeon E3-1231v3.

While both ASan and SoftBound rely on compile-time instrumentation and thereby facilitate native execution of the instrumented program under test, our approach relies on dynamic binary instrumentation by means of execution inside a virtual machine. This forces us to spend around 40 seconds starting up the virtual machine and loading all necessary files into it before the execution of the program under test can begin. Although this must be performed every time a program under test is started in our prototype implementation, this is merely an issue of our implementation that could be alleviated by constantly keeping the virtual machine up and running and starting the program under test multiple times in the same virtual machine instance. Therefore, we disregard this initial startup time and only start measuring the time when execution of the program under test starts.

7.5.2. Results

The mean running times for each configuration are presented in Figure 7.15, Figure 7.16, and Figure 7.17. As expected, our approach incurs a massive performance overhead, slowing down execution by a factor ranging from 5,000 up to 47,000. ASan and SoftBound, on the other hand, incur an overhead that increases the running time by a factor of around five in most cases.

While the overhead of our approach might seem to make our approach entirely unusable in practice, it is important to keep in mind that the primary goal during the development of our prototype was to facilitate an evaluation of how effectively our design can detect OOB writes. Performance, on the other hand, was almost entirely disregarded during development. While this allowed us to create a working prototype within the available time, it also resulted in a number of design decisions with a highly detrimental impact on the overhead.
The most severe such decision is the usage of S2E as a program analysis framework on top of which we implement our approach. While S2E has the potential to achieve excellent performance in some application scenarios due to its hybrid approach to symbolic execution, our use case is not one of them. This is mainly due to the necessity of executing large parts of the program under test in symbolic mode, caused by making pointers symbolic to taint them. As execution in KLEE is notoriously slow, even when there is no need for solving symbolic constraints, this constitutes our primary source of overhead.

Note that our execution time measurements include the time required for setting up the process, i.e., loading the program and shared libraries. As this can be expected to constitute a substantial part of the program's running time when instrumented with ASan and SoftBound but is a comparatively quick endeavor when using our approach, our approach's overhead would likely increase further with more computationally intensive programs.

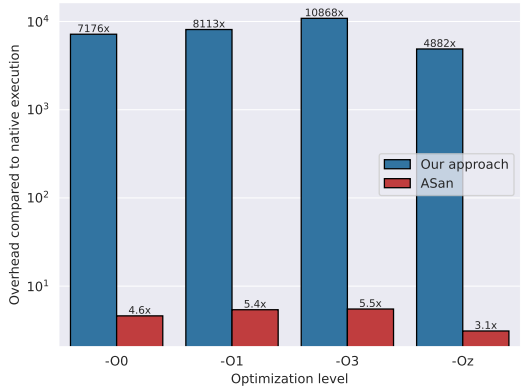


Figure 7.15: Performance overhead for xmllint

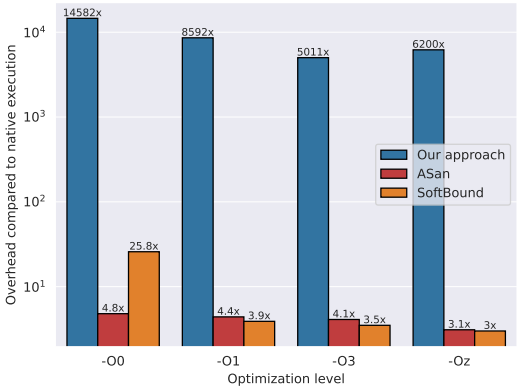


Figure 7.16: Performance overhead for pnm2png

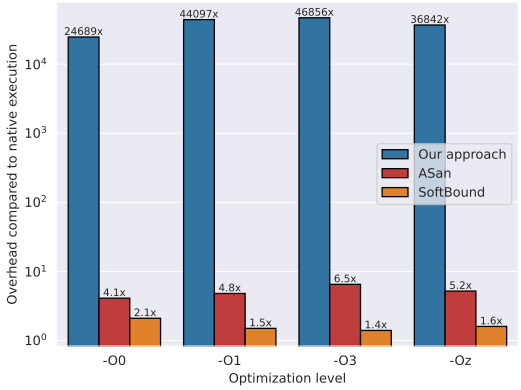


Figure 7.17: Performance overhead for gzip

7.6. Summary

The experiments show that all in all, our approach is well capable of detecting most OOB writes. Our approach successfully detected every OOB write caused by real-world vulnerabilities, thereby exceeding the detection performance of both ASan and SoftBound. Furthermore, while our approach's logic for detecting intra-object OOB writes resulted in a number of false positives, it did not rely on this logic for any of the vulnerabilities we evaluated. Thus, disabling the intra-object OOB write detection logic would cause our approach to fully outperform ASan and SoftBound, with the additional benefit of being entirely non-invasive.

While intra-object OOB write detection remains unsolved by both ASan and SoftBound, our approach's logic for achieving intra-object OOB write detection also turned out to be the most fragile part of its design. Besides the false positives raised for the real-world vulnerabilities, intra-object OOB writes are the only test cases in our testbed experiments for which detection fails. Nevertheless, these experiments show that our approach's detection performance exceeds that of ASan and SoftBound in most ways, with a false positive in the dependent write testbed being the only apparent downside.

The most apparent drawback of our approach is the excessive performance overhead. Although we are convinced that the performance overhead can be decreased by several orders of magnitudes when re-implementing our approach on top of a dynamic taint analysis framework that is not subject to the slowdown caused by symbolic execution in KLEE, no implementation will achieve a performance overhead similar to that of ASan and SoftBound, which allow for native execution of the instrumented program.

Therefore, we can conclude that, in any use case where a low execution slowdown is crucial, e.g., when fuzzing for vulnerabilities, an instrumentation-based approach like ASan and SoftBound should be the tool of choice.
However, when only a small number of executions are necessary, a high recall is desirable, and false positives are tolerable, our approach is superior to ASan and SoftBound. This is especially true if intra-object OOB writes are to be detected. Despite these benefits, it is important to keep in mind that our work's primary goal was to design a *non-invasive* OOB write detection approach that can be used to characterize the effects of vulnerabilities as they exist in programs deployed for production use. Thus, if the analysis of an OOB write vulnerability in such a program is desired, our approach is without an alternative.

8 Discussion

In this chapter, we critically reflect on our work by describing our design, implementation, and evaluation limitations. Furthermore, we identify and outline several possible directions for future work.

8.1. Limitations

A general limitation of our work is the inevitable lack of completeness. As the complexity of modern compilers and optimizations makes it infeasible to identify and account for every possible pattern of generated machine code, we followed an iterative top-down approach in constructing our design. We inspected generated machine code, attempted to identify patterns relevant to our overall goal, and made assumptions to base our design on, followed by attempts to validate these observations and assumptions on more generated code while seeking to construct counterexamples. Thereby, we managed to consistently refine our design at the cost of increased complexity. While this approach is presumably the only feasible way to arrive at a design with satisfying detection performance, it also prevents us from accounting for specific patterns of compiled code that occur only rarely or under particular conditions. However, as we also found a higher degree of completeness to necessitate increased complexity in most cases, a more complete design would likely have been infeasible in this project's scope either way. Furthermore, our evaluation shows that our design can handle a large part of the most common patterns.
While this limitation is not exclusive to our work and concerns most program analysis tools that interact with compiler-generated code, our approach is arguably more severely affected by it. This is a consequence of our non-invasiveness requirement, which, contrary to instrumentation-based sanitizers, prevents our approach from interfering with code generation, thereby fully exposing it to the compiler's caprices.

A further limitation is caused by the fundamental assumption of our approach that, once the bounds attached to a pointer are narrowed to a composite object's child, any derived pointer will require identical or narrower bounds. While this assumption holds in the overwhelming majority of cases, there are nevertheless scenarios in which this assumption causes a legitimate write to be flagged as OOB.
One type of scenario in which this occurs are bounds-shifting instructions, which are the cause for all false positives we encountered in gzip. However, as GEP instructions in the LLVM IR allow identifying occurrences of such bounds shifting operations, our design could be extended to account for such cases.
A further type of scenario in which this assumption is violated are cases in which compiler optimizations cause a pointer to the child of a composite object to be used not only for writing to the child but also to one or more of its siblings. We identified this as the cause for the false positive encountered while evaluating our approach on libpng. However, contrary to bounds-shifting instructions, it is not intuitively clear how such scenarios can be adequately handled. Hence, detecting such false positives would presumably require the usage of heuristics.

A general limitation of our approach's reliance on DWARF debug information for memory layout and type extraction is a strong dependence on the correctness and completeness of this information. Although we did not encounter any occurrences of clearly incorrect memory layout or type information, we discovered several cases in which the information describing the location of variables in memory was incomplete, thereby preventing our approach from identifying all source code-level objects affected by an OOB write. As DWARF debug information has been shown to be faulty in a number of other cases [61], incomplete and potentially even incorrect debug information poses a certain threat to the completeness and thereby usefulness of our approach's results.

A further limitation of our design is disregarding variable-sized objects on the stack. Examples of such objects are variable-length arrays (VLAs) and memory chunks allocated by the `alloca` standard library function. As the size of such objects is only determined at run time, identifying their dimensions requires an approach for intercepting the size parameter at run time and recording it in the memory layout tracker. While this can easily be done for `alloca`

calls by intercepting the corresponding function call, doing so for VLAs requires a more sophisticated approach. Furthermore, intra-object OOB write detection for such variable-sized objects is a significant challenge, as the exact positioning of contained composite objects is only determined at run time. As however, the usage of variable-sized stack objects is frequently discouraged as it prevents certain optimizations from being applied and increases the risk for stack overflows, they are relatively infrequently encountered in real-world software. For this reason, we decided not to support them and treat any variable-length stack object like unused space.

Like any other identity-based OOB write detection approach, our design, by default, cannot perform bounds-checking for writes relative to pointers that are not created by the program itself. This includes pointers returned from syscalls and pointers originating from user input. As such, our approach cannot detect OOB writes caused, e.g., by a format string exploit. However, given our approach's intended use case of characterizing vulnerabilities discovered through fuzzing, this can be considered a minor limitation.

As a consequence of setting our scope to aim for the identification of source code-level objects that are affected by an OOB write, our approach cannot identify most affected lower-level objects. Examples of such objects include spilled registers that do not hold a source code-level variable, stack canaries, and any other object that is stored on the stack but does not correspond to a source code-level object. As overwriting these objects can nevertheless have security implications, both by diminishing and increasing the risk of exploitability, disregarding them in the results prevents us from delivering a complete overview of an OOB write's consequences. However, our evaluation has shown that the stack space used for such objects makes up a relatively small share of the overall stack space.

Lastly, it is important to note that the non-invasiveness we achieve with our approach does not necessarily extend to the temporal domain. Due to the emulation that our implementation relies on, the execution of a program under test not only takes substantially more time than natively, but the temporal run time behavior is likely to become skewed in the way that some parts of the program are slowed down more than others. This potentially causes anomalous behavior in concurrent programs that would not be encountered when executing natively and makes our approach inapplicable for characterizing the effects of OOB writes caused by concurrency issues such as race conditions.

Despite these limitations, we showed that our approach is an effective tool for detecting and analyzing the consequences of OOB writes in C programs and is capable of handling most common code and vulnerability patterns.

8.1.1. Simplifications

To reduce the complexity of our design and implementation, we refrained from incorporating several features that we deemed not to be essential.

One such feature is the consideration of bounds-narrowing and pointer-creating instructions that write the emitted pointer directly to memory. Disregarding such instructions inevitably causes us to miss a certain number of pointer creations and sites at which bounds-narrowing is required, thereby causing unchecked writes or checks with overly wide bounds. However, measuring the occurrence of pointer-creating instructions writing directly to memory revealed that these typically constitute less than 3% of pointer-creating instructions. Similar numbers are expected for bounds-narrowing instructions that write directly to memory.

An additional feature we omitted is the consideration of dynamic bounds-narrowing instructions, i.e., bounds-narrowing instructions for which the target is only determined at compile time. This occasionally causes our approach to perform write checking with overly wide bounds, potentially missing intra-object OOB writes. The frequency with which this occurs heavily depends on the type of program. As shown in Chapter 7, the share of bounds-narrowing instructions that are dynamic can range from less than 5% up to 80%.

A further limitation of our implementation is the simplifying assumption that the lifetime of any object can be described by a single interval, as outlined in Section 6.3.1. To combat the occurrence of overlapping objects, we merge stack objects with the same spatial position, which, however, cannot eliminate all overlapping objects. Thus, a small number of stack objects remains overlapping during dynamic analysis, making it impossible to determine the intended pointee object for some pointers and thereby preventing writes to them from being bounds-checked. Measuring the frequency with which this occurs during the evaluation revealed that this only constitutes a minor problem in most programs but can cause a large number of pointers to remain untainted in other programs, mainly when compiled with aggressive optimizations.

An additional side effect of our practice of merging overlapping variables with identical spatial positions is a slightly reduced interpretability of the results. While we record the source code-defined names of all merged objects in the resulting object, an overwrite of such an object will leave it unclear which of the object names this affects. However, as we found overlapping objects to frequently have the same semantic purpose in the program, the impact on the usefulness of our results is limited.

In our approach for determining the intended destination object of an IR write, described in Section 5.6, any declaration chain-merging declaration encountered while tracing the chain upwards is considered a terminal declaration indicating a dynamic write. We justify this with the inability to determine which of the merged declaration

chains will eventually result in the write being performed. However, this rule is too restrictive when all merged declaration chains originate from the same terminal declaration indicating a static write. Although we never encountered such a case in practice, an entirely sound tracing approach would require tracing beyond any branch-merging declaration and comparing the subjects of the terminal declarations, thereby potentially uncovering additional static writes.

A general limitation of our implementation is that it is limited to performing OOB write detection and characterization for programs loaded into memory at a fixed location known at compile time. For position-independent code, on the other hand, writes cannot be bounds-checked, and memory objects affected by an OOB write within such dynamically loaded sections cannot be identified. This concerns both position-independent executables and dynamically loaded libraries. However, this limitation is merely a matter of implementation we decided to accept to reduce complexity and is not a general limitation of our design.

To simplify call stack tracking and maintaining location information on automatic variables in our implementation, we make two assumptions on the compiler settings with which the program under test is built.
Firstly, we assume that the compiler does not optimize tail calls by reusing the existing stack frame of a function and jumping to the next function instead of calling it.
Secondly, we assume that the program is compiled without frame pointer omission, and automatic variables are therefore always accessed using an offset from the stack frame base pointer `rbp`, instead of from the stack pointer `rsp`.
The compiler performs both tail call optimization and frame pointer omission on higher optimization levels to reduce the number of instructions and increase the number of available general-purpose registers. Therefore, we explicitly disable these optimizations, which slightly weakens our goal of non-invasiveness, as the build process of the program has to undergo minor modifications, and the resulting binary differs slightly. However, the motivation for these two assumptions is merely a reduction of our implementation's complexity and does not constitute an inherent requirement for our design to function.

8.2. Future Work

A continuation and extension of this work in multiple directions is possible and desirable in the future.

An obvious subject for future work is the extension of our design to account for aspects and patterns that we deliberately omitted in this work to reduce complexity. Among others, this includes accounting for dynamic bounds-narrowing instructions and bounds-shifting instructions, the omission of which we observed to have a potentially large impact on our approach's performance for some programs.

A further potential goal of future work is to extend our approach to other instruction set architectures than AMD64, compilers using different intermediate representations than the LLVM IR, and other programming languages - most notably C++.

As shown in Section 7.5, the excessive overhead of our design's implementation effectively prevents its usage in practice. Therefore, it appears worthwhile to investigate possibilities for implementing our design more efficiently, making it scalable and practical to use as a part of other pipelines or in real-world applications. As we identified the execution in KLEE mode and the maintenance of symbolic constraints for tainted pointers to be the most substantial sources of overhead, implementing our approach on top of a dynamic taint analysis framework appears to be a promising way to achieve this.

A further interesting path for future research is designing a system that characterizes OOB writes not only by determining the objects affected by executing a single PoC input but by searching for additional paths and corresponding inputs that trigger the vulnerability, thereby creating a complete profile of its capabilities. This could be achieved by coupling our approach with a directed fuzzer but would require an implementation of our approach with a substantially reduced performance overhead.

Lastly, collecting high-level semantic information directly from a lower-level representation within the compiler backend is a possible way to alleviate the limitations incurred due to low-quality debug information. By extracting object locations, bounds-narrowing instructions, and independent writes at a later stage during compilation, matching can likely be eased, and limitations caused by incomplete DWARF debug information can potentially be eliminated. Therefore, investigating the possibilities for acquiring such information at a late stage during compilation is a promising direction for future work.

9 Conclusion

Over the course of this thesis, we developed an approach for performing non-invasive, instrumentation-less monitoring of compiled C programs to detect OOB writes on the stack and in the global sections and distill their capabilities by identifying affected source code-level objects stored in memory. We implemented our design and evaluated its detection performance on two benchmarks and three real-world vulnerabilities. This showed that our approach can keep up with - and in some cases even exceed - the detection performance of current instrumentation-based state-of-the-art OOB write detection approaches but is more prone to yield false positives.

We can answer our three research questions as follows.

How can out-of-bounds writes within different program sections be detected without invasive modification of the program? We showed that OOB writes in C programs can be detected in a non-invasive manner by combining information acquired from the program's compiler-internal intermediate representation with an analysis of the compiled program and fine-grained monitoring of the program's execution behavior, along with a pointer tainting mechanism.
We found this approach to facilitate an effective detection of inter-object OOB writes, catching all such bugs we evaluated on and causing only a single false positive detection. However, intra-object OOB write detection using this approach performed comparatively poorly and is responsible for most false positives encountered during our evaluation, despite achieving better detection rates than existing state-of-the-art approaches. Altogether, the primary difficulty lies in the fact that, following from the requirement for non-invasiveness, our approach cannot interfere with the compiler's code generation and, as such, must account for every pattern of machine code possibly generated by the compiler, being fully exposed to all of its caprices.

How can the affected regions in memory be mapped to primitive types and composite structures of the source code? By combining information extracted from the DWARF debug information generated by the compiler with our approach for non-invasive OOB write detection, we showed that DWARF debug information constitutes a viable source of information on memory utilization by source code-level objects. It provides detailed information on the position of such objects in most cases, as well as on the exact composition of their types. By maintaining an internal model of the call stack during the program's execution, the location of any source code-level object on the stack and in the global sections can be exactly determined and used for identifying the objects affected by an OOB write from the affected memory addresses. However, we also showed that DWARF debug information fails to rigorously describe all memory usage on the stack, leaving up to 12% of stack space uncharted. In some cases, this is caused by wrongfully omitted variable locations, presumably due to faults in the compiler. In other cases, optimizations cause objects to lack a clear counterpart in the source code.

How does a non-invasive out-of-bounds write detection approach perform at different levels of optimization applied by the compiler? Despite a number of challenges introduced by optimizations, we achieved compatibility with optimized code for our approach by using minor simplifying assumptions. While the usage of any level of compiler optimizations decreased our approach's ability to detect intra-object OOB writes compared to unoptimized code, we found the aggressiveness of employed optimizations to only have a minor impact on this. However, we discovered optimizations to occasionally cause the violation of some of our fundamental assumptions, thereby causing the occurrence of false positives. Lastly, optimizations commonly cause the vulnerability capability profile resulting from our approach to be less concise and more challenging to interpret.

All in all, we can conclude that our approach constitutes a valuable addition to the family of OOB write analysis tools, outperforming existing solutions in terms of recall and compatibility with external code at the cost of an increased number of false positives and a high performance overhead. Most importantly, however, our approach is the first to perform OOB write detection in a fully non-invasive manner. As such, it can not only serve as a valuable aid for human analysts performing bug triaging but may also be used as a building block in a variety of further research directions and tools, ranging from automated bug triaging approaches to machine-assisted and automatic exploit generation.

References

[1] *TIOBE Index for January 2022*, 2022. [Online]. Available: https://web.archive.org/web/20220126205740/https://www.tiobe.com/tiobe-index/ (visited on 01/26/2022).

[2] J. P. Anderson, "Computer Security Technology Planning Study," U.S. Air Force Electronic Systems Division, Tech. Rep., 1972.

[3] *2021 CWE Top 25 Most Dangerous Software Weaknesses*, 2021. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021%5C_cwe%5C_top25.html (visited on 01/11/2022).

[4] ISO Central Secretary, "Programming languages — C," International Organization for Standardization, Geneva, CH, Standard ISO/IEC 9899:2011, 2011.

[5] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2018, ISBN: 1593278284.

[6] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley Professional, 2015, ISBN: 0134190440.

[7] C. Cowan, C. Pu, D. Maier, *et al.*, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proceedings of the 7th conference on USENIX Security Symposium*, 1998.

[8] PaX Team, *Address Space Layout Randomization*, 2001. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt (visited on 01/12/2022).

[9] Z. Y. Ding and C. L. Goues, "An Empirical Study of OSS-Fuzz Bugs," *arXiv*, 2021. eprint: 2103.11518.

[10] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Proceedings of the Network and Distributed System Security Symposium*, 2011. DOI: /10.1145/2560217.2560219.

[11] S. Heelan, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," M.S. thesis, University of Oxford, 2009.

[12] L. Xu, W. Jia, W. Dong, and Y. Li, "Automatic Exploit Generation for Buffer Overflow Vulnerabilities," *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018. DOI: 10.1109/qrs-c.2018.00085.

[13] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," *2012 IEEE Symposium on Security and Privacy*, 2012. DOI: 10.1109/sp.2012.31.

[14] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012. DOI: 10.1109/sere.2012.20.

[15] Meh, *Exim Off-by-one RCE: Exploiting CVE-2018-6789 with Fully Mitigations Bypassing | DEVCORE*, 2018. [Online]. Available: https://devco.re/blog/2018/03/06/exim-off-by-one-RCE-exploiting-CVE-2018-6789-en/ (visited on 12/07/2021).

[16] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*, 2011, ISBN: 9781450306720. DOI: 10.1145/2076732.2076739.

[17] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual Volume 1: Application Programming," Advanced Micro Devices, Tech. Rep. 3.23, 2020.

[18] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," Advanced Micro Devices, Tech. Rep. 3.38, 2021.

[19] H. J. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell, "System V Application Binary Interface: AMD64 Architecture Processor Supplement," Specification, 2018. [Online]. Available: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

[20] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," TIS Committee, Specification, 1995. [Online]. Available: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[21] M. Sebor, *Memory Error Detection Using GCC | Red Hat Developer*, Feb. 2017. [Online]. Available: https://developers.redhat.com/blog/2017/02/22/memory-error-detection-using-gcc (visited on 11/24/2021).

[22] J. Seward and N. Nethercote, “Using Valgrind to detect undefined value errors with bit-precision,” in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A Fast Address Sanity Checker,” in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012. DOI: 10.5555/2342821.2342849.

[24] *Memory safety in the Chromium projects*, 2022. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 03/01/2022).

[25] Adepts of 0xCC, *From theory to practice: analysis and PoC development for CVE-2020-28018 (Use-After-Free in Exim)*, 2021. [Online]. Available: https://adepts.of0x.cc/exim-cve-2020-28018/ (visited on 03/01/2022).

[26] Aleph One, *Smashing the Stack for Fun and Profit*, 1996. [Online]. Available: http://phrack.org/issues/49/14.html (visited on 03/01/2022).

[27] V. v. d. 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 - 15th International Symposium*, 2012. DOI: 10.1007/978-3-642-33338-5_5.

[28] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-Control-Data Attacks Are Realistic Threats,” in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[29] C. Tice, T. Roeder, P. Collingbourne, *et al.*, “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM,” in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[30] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic Generation of Data-Oriented Exploits,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[31] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016. DOI: 10.1109/sp.2016.62.

[32] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005. DOI: 10.1145/1102120.1102165.

[33] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004. DOI: 10.1109/cgo.2004.1281665.

[34] LLVM Project, *LLVM Language Reference Manual — LLVM 13 documentation*, 2021. [Online]. Available: https://releases.llvm.org/13.0.0/docs/LangRef.html (visited on 03/08/2022).

[35] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: a platform for in-vivo multi-path analysis of software systems,” *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS ’11*, 2011. DOI: 10.1145/1950365.1950396.

[36] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[37] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[38] V. Chipounov, V. Kuznetsov, and G. Candea, “The S2E Platform: Design, Implementation, and Applications,” *ACM Transactions on Computer Systems (TOCS)*, 2012. DOI: 10.1145/2110356.2110358.

[39] DWARF Debugging Information Format Committee, “DWARF Debugging Information Format Version 4,” DWARF Standards Committee, Tech. Rep., 2010.

[40] D. Song, J. Lettner, P. Rajasekaran, *et al.*, “SoK: Sanitizing for Security,” *arXiv*, 2018. eprint: 1806.04355.

[41] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009. DOI: 10.1145/1543135.1542504.

[42] The Clang Team, *Hardware-assisted AddressSanitizer Design Documentation*, 2022. [Online]. Available: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html (visited on 01/10/2022).

[43] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, “Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches,” *arXiv*, 2017. eprint: 1702.00719.

[44] T. Kroes, K. Koning, E. v. d. Kouwe, H. Bos, and C. Giuffrida, “Delta Pointers: Buffer Overflow Checks Without the Checks,” *Proceedings of the Thirteenth EuroSys Conference*, 2018. DOI: 10.1145/3190508.3190553.

[45] A. Slowinska, T. Stancescu, and H. Bos, “Body armor for binaries: preventing buffer overflows without recompilation,” *2012 USENIX Annual Technical Conference*, 2012.

[46] *SGCheck: an experimental stack and global array overrun detector*, 2022. [Online]. Available: https://valgrind.org/docs/manual/sg-manual.html (visited on 01/14/2022).

[47] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, “PAriCheck: an efficient pointer arithmetic checker for C programs,” *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’10*, 2010. DOI: 10.1145/1755688.1755707.

[48] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, “Memory Tagging and how it improves C/C++ memory safety,” *arXiv*, 2018. eprint: 1802.09517.

[49] *The Kernel Address Sanitizer (KASAN) — The Linux Kernel documentation*, 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html (visited on 01/10/2022).

[50] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM SIGPLAN Notices*, 2007. DOI: 10.1145/1250734.1250746.

[51] W. Chen, X. Zou, G. Li, and Z. Qian, “KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities,” *USENIX Security Symposium 2020*, 2020.

[52] M. Neugschwandtner, P. M. Comparetti, I. Haller, and H. Bos, “The BORG: Nanoprobing Binaries for Buffer Overreads,” *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015. DOI: 10.1145/2699026.2699098.

[53] Y. Wang, C. Zhang, X. Xiang, *et al.*, “Revery: From Proof-of-Concept to Exploitable,” *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018. DOI: 10.1145/3243734.3243847.

[54] Advanced Micro Devices, “AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions,” Advanced Micro Devices, Tech. Rep. 3.33, 2021.

[55] E. Bendersky, *pyelftools*, 2020. [Online]. Available: https://github.com/eliben/pyelftools.

[56] *Capstone Engine*, 2020. [Online]. Available: https://www.capstone-engine.org/.

[57] *Clang 13 documentation - command guide*, 2021. [Online]. Available: https://releases.llvm.org/13.0.0/tools/clang/docs/CommandGuide/clang.html (visited on 05/01/2022).

[58] H. Rosier, *ripe64*, 2019. [Online]. Available: https://github.com/hrosier/ripe64.

[59] M. Böhme, *oss-security - Invalid writes and reads in libxml2*, 2017. [Online]. Available: https://www.openwall.com/lists/oss-security/2017/05/15/1 (visited on 04/06/2022).

[60] Z. Luo, *stack-buffer-overflow in pnm2png in function get_token*, 2018. [Online]. Available: https://github.com/glennrp/libpng/issues/246 (visited on 05/20/2022).

[61] G. A. D. Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida, and L. Querzoni, “Who’s debugging the debuggers? exposing debug information bugs in optimized binaries,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. DOI: 10.1145/3445814.3446695.

A Appendix

A.1. LLVM IR Analysis Result Format

Entry		Rationale
For each static write	srcLine	Write location in the source code according to debug information, needed for matching to independent write.
	srcColumn	
	srcFileName	
	staticWriteToLocalVar	Whether the write destination is a variable with function scope.
	staticWriteToAutoVar	Whether the write destination is an automatic variable and therefore located on the stack.
	staticWriteDstActualName	The source code-defined name of the write destination. Needed for matching the destination to an object specified by DWARF debug information.
	boundsNarrowingIndices	A list of structure and array indices specifying the bounds-narrowing target, if any. Needed for determining the composite object child to which writes are permitted.
For each BNI	srcLine	Write location in the source code according to debug information, needed for matching to bounds-narrowing ASM instruction.
	srcColumn	
	srcFileName	
	boundsNarrowingIndices	A list of structure and array indices specifying the bounds-narrowing target, if any. Needed for determining the composite object child to which bounds are to be narrowed.
	typeMnemonic	A textual identifier of the type to which the bounds-narrowing instruction expects a pointer. Used for comparing the expected type with the actual type indicated by the pointer taint during dynamic analysis.

Table A.1: Information resulting from the LLVM analysis pass, to be ingested during the ELF analysis

A.2. Example of Analysis Results


```
{"function": "xmlSnprintfElementContent",
"instruction": "0x464c49",
"pruned_call_stack": false,
"affected_ranges": [
   {"overwrite_lower": "0x7ffedd455fb8","overwrite_upper": "0x7ffedd456265",
    "affected_stack_frames": [
      {"function": "xmlValidateElementContent",
        "stack_frame_bottom": "0x7ffedd45601f",
        "stack_frame_top": "0x7ffedd453890",
        "overwrite_lower_rel": "RBP-88",
        "overwrite_upper_rel": "RBP+15",
        "affected_objects": [
          {"name": "cont", "addr": "RBP-88", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "parent", "addr": "RBP-80", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "name", "addr": "RBP-72", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "merged<cur,child>", "addr": "RBP-64", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "ret", "addr": "RBP-56", "size": 4, "first_ow_byte": 0, "last_ow_byte": 3},
          {"name": "<unknown/padding>", "addr": "RBP-52", "size": 4, "first_ow_byte": 0, "last_ow_byte": 3},
          {"name": "exec", "addr": "RBP-48", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "<unknown/padding>", "addr": "RBP-40", "size": 40, "first_ow_byte": 0, "last_ow_byte": 39},
          {"name": "<saved RBP>", "addr": "RBP", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "<saved RIP>", "addr": "RBP+8", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7}]},
      {"function": "xmlValidateOneElement",
        "stack_frame_bottom": "0x7ffedd4560cf","stack_frame_top": "0x7ffedd456020",
        "overwrite_lower_rel": "RBP-160","overwrite_upper_rel": "RBP+15",
        "affected_objects": [
          {"name": "<unknown/padding>", "addr": "RBP-160", "size": 16, "first_ow_byte": 0, "last_ow_byte": 15},
          {"name": "fn", "addr": "RBP-144", "size": 50, "first_ow_byte": 0, "last_ow_byte": 49},
          {"name": "<unknown/padding>", "addr": "RBP-94", "size": 22, "first_ow_byte": 0, "last_ow_byte": 21},
          {"name": "name", "addr": "RBP-72", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "<unknown/padding>", "addr": "RBP-64", "size": 4, "first_ow_byte": 0, "last_ow_byte": 3},
          {"name": "extsubset", "addr": "RBP-60", "size": 4, "first_ow_byte": 0, "last_ow_byte": 3},
          {"name": "ret", "addr": "RBP-56", "size": 4, "first_ow_byte": 0, "last_ow_byte": 3},
          {"name": "<unknown/padding>", "addr": "RBP-52", "size": 52, "first_ow_byte": 0, "last_ow_byte": 51},
          {"name": "<saved RBP>", "addr": "RBP", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "<saved RIP>", "addr": "RBP+8", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7}]},
      {"function": "xmlSAX2EndElementNs",
        "stack_frame_bottom": "0x7ffedd45611f", "stack_frame_top": "0x7ffedd4560d0",
        "overwrite_lower_rel": "RBP-64", "overwrite_upper_rel": "RBP+15",
        "affected_objects": [
          {"name": "<unknown/padding>", "addr": "RBP-64", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "node_info", "addr": "RBP-56",
            "affected_members": [
              {"name": "node", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
              {"name": "begin_pos", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
              {"name": "begin_line", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
              {"name": "end_pos", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
              { "name": "end_line", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7}]},
          {"name": "<unknown/padding>", "addr": "RBP-16", "size": 16, "first_ow_byte": 0, "last_ow_byte": 15},
          {"name": "<saved RBP>", "addr": "RBP", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7},
          {"name": "<saved RIP>", "addr": "RBP+8", "size": 8, "first_ow_byte": 0, "last_ow_byte": 7}]},
      {"<4 stack frames omitted>"}
      {"function": "xmlCtxtReadFile",
        "stack_frame_bottom": "0x7ffedd45626f", "stack_frame_top": "0x7ffedd456230",
        "overwrite_lower_rel": "RBP-48", "overwrite_upper_rel": "RBP+4",
        "affected_objects": [
          {"name": "<unknown/padding>", "addr": "RBP-48", "size": 48, "first_ow_byte": 0, "last_ow_byte": 47},
          {"name": "<saved RBP>", "addr": "RBP", "size": 8, "first_ow_byte": 0, "last_ow_byte": 4}]}]}]},
{"function": "xmlSnprintfElementContent",
 "instruction": "0x464d47",
 "pruned_call_stack": false,
 "affected_ranges": [
  {"overwrite_lower": "0x7ffedd456266","overwrite_upper":  "0x7ffedd456266",
 "affected_stack_frames": [
  {"function": "xmlCtxtReadFile",
   "stack_frame_bottom": "0x7ffedd45626f","stack_frame_top":  "0x7ffedd456230",
   "overwrite_lower_rel": "RBP+5","overwrite_upper_rel": "RBP+5",
   "affected_objects": [
     {"name": "<saved RBP>", "addr": "RBP", "size": 8,  "first_ow_byte": 5, "last_ow_byte": 5}]}]}]}
```


Figure A.1: Truncated analysis results for CVE-2017-9047 PoC when compiling libxml with `-O1`