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

Linus Hafkemeyer[1], Jerre Starink[2], Andrea Continella[2]

[1] TU Delft [2] University of Twente
`linus.hafkemeyer@pm.me`, {`j.a.l.starink,a.continella`}`@utwente.nl`

Abstract. Despite the high level of automation that fuzzing has brought into the vulnerability research process, the assessment of a discovered vulnerability's implications mostly requires human expertise and intuition. A promising approach to reduce such a manual effort is the automatic extraction of vulnerability characteristics that provide vital clues for exploitability. In this work, we focus on out-of-bounds write vulnerabilities and investigate how to automatically distill the set of source code-level objects affected by such unintended writes. As this poses unique challenges with regard to the invasiveness of the analysis methods, we propose a novel approach that enables monitoring a compiled program for spatial memory safety violations *without* the need for heavy instrumentation. We implement DIVAK, a prototype of our design, and we evaluate it on both benchmarks and real-world vulnerabilities, showing that its detection and characterization capabilities outperform instrumentation-based tools in several scenarios, at the cost of an increased overhead.

Keywords: Vulnerability Analysis · Out-of-bounds Writes.

1 Introduction

Out-of-bounds (OOB) writes [1] are still regarded as one of the most dangerous types of software vulnerabilities [22]. Over the years, a vast number of defenses against OOB writes and other memory corruption bugs have been proposed. Preventive approaches such as the deprecation of unsafe functions [16] and memory-safe languages [19,12] reduce the risk but cannot solve the problem without full adoption. Mitigation approaches introduced into operating systems and compilers complicate or prevent exploitation [9,27], but attackers continue to find ways for evading them. Detection approaches based on static and dynamic analysis have been hugely successful [11]. However, only a relatively small share of discovered bugs has relevant security implications. Thus, further investigation into the severity of discovered bugs and 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 long time. Therefore, further automating the process that follows the discovery of a bug can substantially decrease the time required to develop a patch. Unfortunately, this process is often an intricate task that largely relies on human expertise and intuition, making full automation difficult.

Recent research [32] has shown that *partial* automation is a promising alternative to approaches based on fully Automatic Exploit Generation (AEG) [3,5,14,15,40]. By automatically distilling characteristics of a vulnerability that are decisive for its exploitability, experts can base their assessment of the bug's security implications on a concise high-level summary, accelerating the triaging process. Besides, AEG based on human-interpretable vulnerability characteristics can make intermediate results substantially more helpful for manual analysis.

We study how to automatically distill such characteristics of OOB writes from programs. In contrast to the state-of-the-art, we aim to extract capabilities that are truthful to the form in which the program under test is deployed in practice, thus revealing vulnerability characteristics that are relevant *not only in laboratory or debugging settings* but for the program's *real-world usage*. We realize this by developing a system that takes a program under test together with an input that is suspected of causing an OOB write and dynamically performs fine-grained monitoring for OOB writes, mapping affected memory regions to the corresponding source code-level entities and reporting the results in a concise and easily interpretable form. Our system is meant to assist security analysts in triaging potential OOB write vulnerabilities, automating their identification and capability extraction phase.

Designing such a system comes with three main challenges: (1) Many approaches are invasive due to heavy instrumentation, modifying the memory layout and runtime behavior of the program and thus making insights inapplicable to the original program; (2) The compilation to machine code causes large parts of source code semantics to be lost, including 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 information is vital for detecting OOB writes, and is essential for achieving easy interpretability of the results; (3) Modern compilers perform optimizations during compilation to increase the program's efficiency, which often causes substantial modifications to the program's machine code-level structure and memory layout.

To address such challenges, we propose a new approach for the dynamic characterization of OOB write vulnerabilities in C programs. Contrary to existing works, our approach does not modify the program through instrumentation and, as such, is entirely *non-invasive*. Instead, we provide a conceptual framework for making source code-level semantic information available to our low-level OOB write-checking technique. As the issue of invasiveness predominantly concerns the stack and global sections, we only focus on the characterization of OOB write vulnerabilities within these regions of programs written in C, compiled with Clang for Linux on AMD64 platforms, and leave out heap-based OOB writes from our analysis. We implement our approach in a system named DIVAK, which achieves a detection rate of 89% on the RIPE benchmark [39], compared to the 70% and 34% achieved by the instrumentation-based current state-of-the-art approaches ASan and SoftBound—at the cost of an increased execution time overhead, along with a slightly increased chance of false positives. Ultimately, DIVAK precisely highlights the source code-level objects affected by OOB writes, assisting humans in triaging potential vulnerabilities.

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

```
void login(struct userEntry* userPtr) {
  struct userEntry user = *userPtr;
  char realPw[32], tryPw[32]; bool pwOk;
  getUserPassword(&user, realPw);
  fgets(tryPw, 32, stdin);
  if (pwOk = !strcmp(realPw, tryPw))
    setUserLoggedIn(&user);
}
```

Fig. 1: Motivating example.

Contributions. We make the following contributions:

– We introduce a technique for low-level bounds-checking by leveraging the intermediate program representation during compilation, overcoming the lack of source code-level semantic information.
– We design a non-invasive OOB write characterization approach able to triage spatial memory safety violations on the stack and in the global sections.
– We implement Divak and we evaluate it on artificial benchmarks and real-world vulnerabilities, showing its advantages over state-of-the-art tools.

We make our dataset and code available: https://github.com/utwente-scs/divak.

2 Motivation

No existing tool for detecting OOB writes is suitable for characterizing their capabilities and identifying their source code-level consequences. In fact, for our scenario, i.e., triaging potential vulnerabilities in real conditions, all publicly available approaches suffer from one or more of the following limitations.

Inability to detect intra-object OOB writes. Many approaches cannot detect intra-object OOB writes within composite objects such as structures [29,8,31,37,20,41]. However, intra-object OOB writes are well capable of inducing security issues and need proper triaging, as is illustrated in Figure 1. Here, an overflow of `username` can corrupt the `isAdmin` flag, enabling a non-control data attack. Thus, their inclusion in a vulnerability's capability profile is critical.

Required hardware support. Some approaches [26,30,8] rely on extensions to the ISA of the CPU to perform OOB write detection. While such ISA extensions are available for SPARC [30], ARM64 [8], and some historic Intel AMD64 CPUs [26], they are not included in the ISA of any recent AMD64 CPUs.

Invasive modification of the program. Existing approaches significantly affect the program's memory layout due to their instrumentation. Such modifications fall into the following categories: (1) Introduction of poisoned red zones around memory objects; (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. Consider the snippet shown in Figure 1 and its stack layout as implemented by ASan [29] and SoftBound [23] in Figure 2. We can clearly see both solutions heavily modify the stack frame layout. This, accompanied with the extra register spilling introduced by the checking logic as well as compiler optimizations on the instrumented program, makes reliably identifying and triaging the memory objects affected by OOB writes within the non-instrumented program challenging.

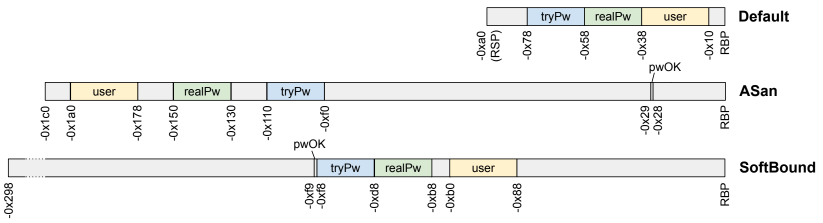

Fig. 2: Stack layouts of the function in Figure 1 (Default, ASan, and SoftBound).

Table 1: Our approach vs. existing memory sanitizers.

	ASan [29]	HWAsan [8]	Memcheck [31]	SGcheck [37]	SoftBound [23]	Delta Ptrs [20]	Intel MPX [26]	BinArmor [33]	PAriCheck [41]	Our approach
OOB writes detection in globals	✓	✓	×	✓	✓	✓	✓	✓	✓	✓
OOB writes detection on stack	✓	✓	×	✓	✓	✓	✓	✓	✓	✓
OOB writes detection 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 mem. 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

*: 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

Instability after Out-of-bounds write. Most tools terminate the program's execution after detecting one OOB write. While this behavior can often be circumvented, it might cause the program to follow invalid execution paths for the non-instrumented program. Consider a detection approach that uses the stack to store metadata such as bounds information. An OOB write that overwrites this metadata may result in false positives and negatives or let the program crash. As we wish to reliably triage OOB writes during the execution of a program, this limitation makes most existing approaches unusable for our case.

3 Divak: Design

To overcome the challenges described in Section 2, we design Divak, a pointer-based OOB write detection approach that characterizes spatial memory violations on the stack and in the global sections in a deterministic fashion. By not interfering with the compiled machine code, Divak is entirely non-invasive and does not rely on special hardware support. As such, differently from existing work (Table 1), any insights about the effects of OOB write vulnerabilities in the examined binary also hold when the program is not monitored by our approach.

Our approach categorizes memory objects for bounds-checking. For *composite objects* like structs and arrays of structs, we consider the inner structure for the detection of *intra-object* OOB writes. Any other object is instead a *unitary object*—a homogeneous chunk of memory for which we disregard inner structure.

We focus on detecting OOB writes occurring in the static sections `.data` and `.bss`, as well as on the stack. We disregard heap-based OOB writes, as their characteristics are not necessarily distorted by instrumentation-based detection approaches—e.g., SoftBound neither allocates heap memory nor affects the allocator. Thus, Divak's main novelty, i.e., its non-invasiveness, is not essential in this scenario. Nevertheless, Divak could be extended to support heap-based OOB writes with little engineering effort, and it is compatible with existing tools that target the heap [13,29]. Besides, we assume that target programs are compiled without frame pointer omission and tail call optimization. Finally, as the majority of memory-modifying instructions that can potentially cause OOB writes are those of the `mov` family, we focus our bounds-checking on this family.

3.1 Approach Overview

Our approach takes as input the source code of a program and a proof-of-concept (PoC) input suspected of causing an OOB write and it outputs information about the effects of all discovered OOB writes on source code-level objects. At a high level, we perform three phases: preliminary analysis, static analysis, and dynamic analysis. In the first phase, we instrument the compilation phase of the target program to passively collect information about debug symbols, data structures, and write operations. This information enables us to map properties of source code to compiled code, which allows us to later pinpoint the specific source code-level objects affected by OOB writes. Besides, it also allows us to handle the loss of semantics caused by the compilation process, which is critical to characterize OOB writes *without* requiring invasive modifications of the program.

In the second phase, leveraging the collected information at compile time, we statically analyze the target binary to identify variables and parameters stored on the stack or in the globals, determine their sizes, identify pointer-creating instructions, and determine the destination objects of write operations. Besides, in this phase, we determine the internal structure of composite types such as `struct`, which is essential to detect intra-object OOB writes.

In the third phase, we dynamically analyze the target program by using the findings obtained through static analysis, taint pointers, and identify write operations that have an effect beyond the boundaries of the intended destination objects (IDOs). Here, we map our results back to the source code domain using the information we collect in the previous phases. Finally, because we do not alter the state or memory layout of the program at run-time, our approach guarantees that, by design, execution continues reliably after detecting an OOB write.

Although the goals of preliminary and static analysis could theoretically be achieved by modifying the compiler, this would come with several drawbacks: (1) Heavy modifications of highly complex code at multiple compilation stages with little documentation; (2) Potentially altered binaries due to modifications, including in production builds; (3) Incompatibility with custom or new optimization passes. Therefore, we opted for the more portable hybrid approach.

Our design for detecting OOB writes relies on the identification and special treatment of the following types of machine code instructions.

Independent writes. For independent writes in the machine code, the dominant component from which the destination address is computed in the operand is either given by an immediate value or a stack frame boundary register (`rbp` or `rsp`). This has two important implications. First, independent writes can only write to global objects or within the stack frame of the containing function. Second, their intended destination object does not change at runtime. An example of an independent write is the instruction `mov [rsp + rax], cl`, which may access an array on the stack. Here, `rsp` constitutes the dominant component of the address calculation as its value will be substantially larger than the value in `rax`. Now, `rsp` being a stack boundary register makes this an independent write.

Dependent writes. For dependent writes, the dominant component used to compute the destination address is given by a general-purpose register as opposed to a stack frame boundary register. Thus, dependent writes rely on a previous instruction for determining the pointer used as the basis of the address computation. This requires detaching the logic for determining the intended destination object from the logic for checking the legality of the write. An example of a dependent write is the instruction `mov [rcx + rax*8], rdx`, which may access an array based at the address specified by `rcx`. Here, the dominant component is given by a general-purpose register, making this a dependent write.

Pointer-creating instructions (PCIs). To facilitate bounds-checking of dependent writes, it is essential to taint pointers with their intended pointee object as early as possible by identifying the instructions at which pointers are created.

Bounds-narrowing instructions (BNIs). Children of composite objects are often accessed by offsetting a pointer to the object. To detect intra-object OOB writes, it is therefore essential to adjust a pointer's bounds information as soon as it starts pointing to a child object. While PCIs create a new pointer, BNIs transform an existing pointer to a pointer referencing the original object's child.

Algorithm 1 shows a high-level overview of our approach's dynamic analysis stage, and is intended to be applied to every instruction of the program upon its execution. Operations related to the core challenges solved by our design are marked in orange. For a dependent write, we identify the intended destination object from the destination pointer's taint. Using the bounds-narrowing information associated with the dependent write, we check if a write is fully in-bounds. For independent writes, the intended destination object is fixed at compile time, thus we check if the write is in bounds from the knowledge of the written bytes and the exact instruction. For PCIs and BNIs, we taint newly created pointers and re-taint existing ones to narrow their bounds according to the new pointee.

3.2 Memory Layout Extraction

Maintaining information on which objects occupy which memory regions during dynamic analysis is essential for detecting OOB writes and mapping the affected regions to their corresponding objects. To do so, we leverage DWARF debug data during static analysis. While the locations of global objects are generally

Algorithm 1 High-level pseudocode description of our dynamic analysis stage.

```
1: if $inst$ is write then
2:     $dstAddr \leftarrow$ getWriteDstAddr($inst$)
3:     $nBytes \leftarrow$ getWriteBytesNum($inst$)
4:     if $inst$ is dependent write then
5:         $taint \leftarrow$ getPointerTaint($dstAddr$)
6:         $ido \leftarrow$ getObjectFromTaint($taint$)
7:         if $inst$ is BNI then
8:             $bniTarget \leftarrow$ getBoundsNarrowingTarget($inst$)
9:             $ido \leftarrow$ narrowObject($ido, bniTarget$)
10:     else
11:         $ido \leftarrow$ getObjectFromIndependentWrite($inst$)
12:     if [$dstAddr$, $nBytes$ - 1] is not in [$ido.start$, $ido.end$] then
13:         reportOOBw()
14: else if $inst$ is PCI or BNI then
15:     $bniTarget \leftarrow$ getBoundsNarrowingTarget($inst$)
16:     $ptr \leftarrow$ getResultingPointer($inst$)
17:     $ido \leftarrow$ getPointeeObject($ptr, bniTarget$)
18:     taintPointer($ptr, ido$)
```

specified by a fixed address, stack objects are referenced as offsets from a stack frame register. We track the program's call stack at runtime by determining the start addresses of functions and monitoring for `call` and `ret` instructions.

Compiler optimizations, which typically decrease the number of objects stored on the stack, frequently reduce the lifetime of objects in memory to one or more instruction address intervals, using the space for different purposes during the remaining part of the function. As DWARF provides detailed information on the lifetimes of objects, we leverage this to record the location of objects not only in a spatial but also in a temporal dimension.

3.3 Intended Destination Objects Identification

For bounds-checking independent writes, we rely on the fact that their intended destination object is fixed at compile time. As high-level semantics are lost during compilation, we implement bounds-checking as a final LLVM IR analysis pass, before the translation into machine code. By leveraging the IR, we avoid directly matching independent writes to memory objects and instead take a detour as follows: (1) Identify independent writes in the IR and determine their destination variable; (2) Match each independent write in ASM to its corresponding independent write in the IR; (3) Match each destination IR variable to its corresponding object in memory.

Determining independent IR writes and destination variables. For identifying independent writes in the IR during the preliminary analysis, we focus on three typical representations of `mov`-family instructions in the IR: the `store` instruction, and the `llvm.memcpy` and `llvm.memset` intrinsics. To test whether an IR write is independent and to find the variable it modifies, we trace back its def-use chain. Starting from the write's operand that specifies the destination, we find the definition that created this pointer and repeat this procedure until there are no more unambiguous predecessors. If we end up at a local or global variable definition, we conclude the write is independent. If we encounter a BNI in the def-use chain, we keep track of the child to which the pointer is modified.

Matching independent writes from ASM to IR. To find the corresponding independent IR write of each independent ASM write during static analysis, we rely on line number debug information that maps instructions to the source file, line, and column at which the corresponding source code is located. While the conveyed location information is irrelevant to us, we can use its distinctive features to map write instructions in the machine code to the IR domain. In practice, however, this mapping is rarely bijective. This is caused by the inherent differences between AMD64 assembly and the IR and our disregard for certain memory-modifying instructions in ASM and the IR. As such, this constitutes a best-effort approach that occasionally fails to match an independent write.

Matching IR variables to memory objects. Determining the destination object for independent IR writes is arguably the simplest step as we can match on the variable names during the preliminary analysis. If bounds-narrowing traversable definitions are encountered in the write's def-use chain, and the intended destination object is thus a child of a composite object, we leverage the information obtained while handling the BNI to determine this child.

3.4 Pointer-Creating Instructions Identification

According to our experience, two types of instructions are responsible for virtually all pointer creations in machine code generated by Clang. First, instructions such as `lea rax, [rbp + 8*rbx - 72]` combine several arithmetic operations and registers to compute an address. This causes the `lea` instruction to be used for almost all cases in which a pointer relative to a stack frame boundary register is created, as it occurs when creating a pointer to an automatic variable. Second, `mov` instructions with a source address into a static section (e.g., `mov edi, 0x409678`) are typically emitted when a pointer to a static variable is created.

3.5 Bounds-Narrowing Instructions Detection

Within compiled code, identifying the locations where bounds-narrowing is performed is challenging. For this, we again leverage the LLVM IR, where pointer manipulation is performed with `getelementptr` (GEP) instructions. For each of them, we test during the preliminary analysis whether the instruction transforms a composite object pointer into one of its (recursive) children. If so, we determine the narrowed child from the instruction. We hereafter refer to this as the *bounds-narrowing target* (BNT).

In compiled code, three types of instructions are, according to our experience, potential candidates for BNIs that are relevant to our goal of bounds-checking writes. First, `add` instructions, which are frequently used for offsetting pointers to struct fields and can be bounds-narrowing if the incremented number is stored in a general-purpose register capable of holding a pointer. Second, `lea` instructions, which we also previously identified as pointer-creating. As the creation of a pointer to a composite object's child requires bounds-narrowing to be performed immediately upon pointer creation, their consideration is essential. Third, the `mov` family, which can act as BNIs in three ways: (1) If it is a PCI to the child of a composite static variable; (2) If a pointer to the beginning of the first child of a composite object is created from a pointer to the composite object; (3) If it writes to the child of a composite object, the address to which is only calculated within the destination operand. Here, the narrowed pointer is used immediately for writing and discarded afterwards. To match BNIs in the IR to their compiled counterparts, we again leverage line number debug information.

3.6 Intended Pointee Objects Determination

Determining a pointer's intended pointee object (IPO) during dynamic analysis is essential for bounds-checking dependent writes. We do so by leveraging memory layout information from debug metadata and bounds-narrowing targets identified during the IR analysis. The need of determining an IPO arises at four different types of instructions in our design: bounds-narrowing dependent writes, bounds-narrowing PCIs, ordinary BNIs, and ordinary PCIs. For each type, the required actions, depending on if the pointer is tainted, are shown in Table 2.

4 Implementation

We implemented Divak in 2,700 SLOC of C++ and 1,900 SLOC of Python code on top of the S2E in-vivo symbolic execution platform [7] and an LLVM pass (Figure 3). The choice of S2E as an analysis platform was mainly motivated by its facilitation of quick prototyping in this case, minimizing the development effort for the non-core functionality of Divak. Furthermore, its symbolic execution capabilities provide us with the ability to perform taint analysis with effectively infinitely many taint colors by maintaining a mapping from symbolic values' internal identifiers to pointee objects. As a negative side-effect, S2E introduces significant performance overhead, which indicates that it may not be the ideal choice in real-world applications. We discuss this aspect in Section 6.

Preliminary analysis. Divak's first stage concerns the compilation using Clang 13.0.1 and our analysis of the LLVM IR. By doing so, we get the required analysis results without modifying the program, thereby achieving non-invasiveness.

To reduce complexity, we disregard two types of BNIs: *(1) Dynamic BNIs* are those BNIs for which the (recursive) child to which a pointer is created is only determined at run time. While omitting them can cause intra-object OOB writes to remain undetected, they typically only occur when arrays of structs

Table 2: Approach for determining the intended pointee object of a pointer in different scenarios. DW=Dependent Write.

	Pointer is tainted	**Pointer is not tainted**
BNI+ DW	Narrow pointer bounds according to target and use immediately to check write.	Previously missed pointer tainting. Cannot identify IPO, hence cannot narrow bounds.
BNI+ PCI	Not actually a PCI, treat like BNI.	Determine pointee from BNT and memory layout, taint pointer.
BNI	Narrow pointer bounds according to target and re-taint.	Previously missed pointer tainting. Treat like BNI + PCI.
PCI	Is a BNI but was not matched. Cannot narrow bounds.	Determine pointee from memory layout, taint pointer.

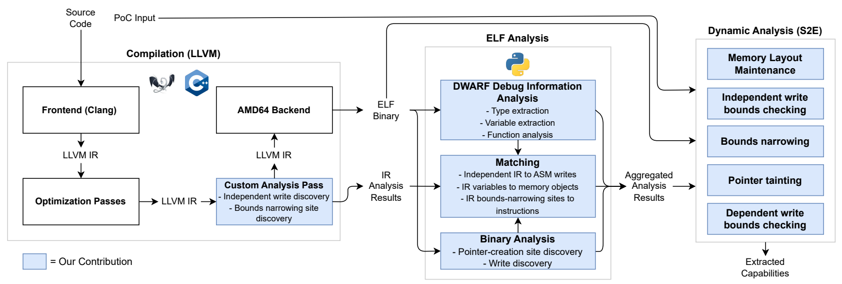

Fig. 3: Schematic diagram of DIVAK, the pipeline implementing our design.

are indexed. *(2) Bounds-shifting instructions* serve to calculate the pointer to a sibling element in the same array, thus violating our assumption that a pointer's bounds can be narrowed by a BNI but can never be widened again. Omitting them can potentially cause legitimate dependent writes to be reported as OOB.

To alleviate poor line number debug information arising from optimizations, we attach random synthetic line numbers to independent IR writes and BNIs that are missing line numbers in the LLVM IR.

Static analysis. The second stage in our pipeline analyzes the compiled binary and combines the obtained results with the high-level semantic information extracted from the LLVM IR. We identify all variables and formal parameters stored on the stack or in the globals from the DWARF debug information to facilitate memory layout tracking, recording their location, lifetime, and type. We consider the smallest interval encompassing all DWARF-specified lifetime intervals as the object's lifetime. Besides, we determine the sizes of all specified types, as well as the inner structure of composite types. To address the challenges arising from representing memory lifetimes as a single interval, we merge objects with identical spatial dimensions whose lifetimes overlap.

To find pointer-creating instructions and independent ASM writes, we identify the instructions satisfying our definitions (Section 3). We then obtain their debug information and find their corresponding independent IR writes with identical files, lines, and column information. By finding the DWARF-specified object corresponding to each independent IR write's destination object through name-based matching, we identify the bounds of each matched independent ASM write's destination object and ease bounds-checking during dynamic analysis.

Dynamic analysis. DIVAK runs and monitors the target program implementing Algorithm 1 in three plugins for S2E [7]. We distinguish dependent and independent writes based on their destination address. If it is symbolic, the write is dependent, and we obtain the intended destination object from the pointer's taint to test if all written bytes are within the object's interval. If it is concrete and we statically identified an independent write at this location, we check if all written bytes are part of the independent write's intended destination object.

We register callbacks that are invoked when executing instructions identified as a PCI or BNI during static analysis. When a PCI or BNI is executed, it is handled as described in Section 3. We determine the new intended pointee object based on the taint of the pointer, the instruction type, the bounds-narrowing in-

Table 3: Detection performance of DIVAK for dependent OOB writes in RIPE testbed at different optimization levels.

		memcpy				strcpy				strncpy				sprintf				snprintf				strcat				strncat				sscanf				fscanf				homebrew			
		0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z
stack	ret	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	baseptr	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩*	🟩*	🟩
	funcptr (var)	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	funcptr (param)	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩
	structfuncptr	🟩	🟧	🟧	🟧	🟩	🟧	🟧	🟧	🟩	🟧	🟧	🟧	🟩*	🟦	🟦	🟦	🟩	🟧	🟧	🟧	🟩*	🟧	🟧	🟧	🟩	🟧	🟧	🟧	🟩	🟧	🟧	🟧	🟩	🟧	🟧	🟧	🟩	🟧	🟧	🟧
	longjmp (var)	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	longjmp (param)	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩
bss	funcptr	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	structfuncptr	🟩*	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟧	🟩	🟩	🟩	🟧	🟩*	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩
	longjmp	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
data	funcptr	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	structfuncptr	🟩	🟧	🟩*	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩*	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩
	longjmp	🟩	🟩	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩*

🟩=detected, 🟧=not detected, 🟦=not possible, ∗=manually validated results

formation, as well as the type of the old intended pointee object. We then taint the pointer accordingly. We refrain from tainting pointers that are not stored in a register but immediately written to memory. To further improve performance, we intercept function calls to several standard library functions such as `memcpy` and perform premature bounds-checking if the destination pointer is tainted. Afterward, we untaint all function arguments to allow for faster emulated execution of the library function. Finally, we group OOB writes occurring at the same instruction and identical call stack, merge the overwritten intervals, identify affected source code-level objects, and report our results in a JSON format.

5 Evaluation

We evaluate DIVAK's performance in terms of OOB write detection efficacy both under laboratory conditions and in real-world programs, as well as performance overhead, and we compare it with ASan and SoftBound.

5.1 Dependent OOB Writes Detection

To measure DIVAK's ability to detect dependent OOB writes, we use a subset of a 64-bit version [28] of the RIPE testbed [39]—designed to test defenses against buffer overflow exploits. By varying the location, the type of overwritten pointer, and the function causing the OOB write, we obtain 122 test cases. By the design of RIPE, some parameter combinations are not possible. We conduct a manual best-effort validation of the affected memory objects identified by DIVAK by comparing our results with the DWARF memory layout, the memory layout to be expected from the source code, and by inspecting the disassembled code.

Results. DIVAK achieves a detection rate of 89%, as shown in Table 3. Failing test cases are limited to intra-object OOB writes under optimizations, with incomplete line number debug information for the corresponding BNI as the root cause. Manually validating DIVAK's output yields flawless results for OOB writes

Table 4: Detection performance of ASan for dependent OOB writes in RIPE testbed at several optimization levels.

		memcpy				strcpy				strncpy				sprintf				snprintf				strcat				strncat				sscanf				fscanf				homebrew			
		0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z
stack	ret	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩
	baseptr	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟩
	funcptr (var)	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	funcptr (param)	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	structfuncptr	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧
	longjmp (var)	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	longjmp (param)	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩
bss	funcptr	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	structfuncptr	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧
	longjmp	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
data	funcptr	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	structfuncptr	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧
	longjmp	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

🟩=detected, 🟧=not detected, 🟦=not possible

in the global sections. On the stack, overwritten ranges are correctly identified. However, with optimizations enabled, the high-level counterparts of roughly 30% of the affected stack objects are not identified due to incomplete DWARF data.

ASan achieves a detection rate of 70%, as shown in Table 4. From our results, it is clear that ASan's primary drawback lies in the inability to detect intra-object OOB writes. While they were detected for `sscanf` and `fscanf`, manual analysis suggests this is caused by a bug in the testbed. SoftBound detects 34% of OOB writes. Similarly, SoftBound is limited by the inability to detect intra-object OOB writes and its reliance on six unimplemented wrapper functions.

5.2 Independent OOB Writes Detection

We evaluate DIVAK's independent OOB write detection performance using a testbed we designed for this purpose and release along with our code. The testbed comprises four parameter dimensions with a total of 44 test cases and largely reproduces the vulnerability scenarios of RIPE using independent writes.

Results. DIVAK successfully detects the OOB write in 95% of the test cases (Table 5). 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. In these cases, the fault occurs in the IR analysis, where tracing the write's declaration chain terminates prematurely. Specifically, the source pointer of a bounds-narrowing `getelementptr` 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 the analysis to conclude that this is not a relevant BNI, as the subject is not an object we consider composite.

Investigating the series of false positives raised by DIVAK reveals a violation of our assumption that pointers point to their intended destination object upon creation. When compiled with `-O1`, a pointer to a stack object is created using a `lea` but only offset to its intended pointee object by an `add` that immediately

Table 5: Detection of independent OOB writes in our testbed. OOB writes occur in isolation (iso) or in a function containing further unrelated writes (svd).

			Divak								ASan								SoftBound							
			iso				svd				iso				svd				iso				svd			
			0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z	0	1	3	z
continuous	stack	ret	🟩	🟩*	🟩	🟩	🟩*	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		baseptr	🟩	🟩	🟩	🟩	🟩	🟨*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		funcptr	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		structfuncptr	🟩	🟩	🟧	🟩	🟩*	🟨	🟧	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
		longjmp	🟩	🟩	🟩	🟩	🟩	🟨	🟩*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	data	funcptr	🟩	🟩	🟩*	🟩	🟩	🟨*	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		structfuncptr	🟩	🟧	🟩	🟧	🟩	🟥	🟩*	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
		longjmp	🟩	🟩*	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	bss	funcptr	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		structfuncptr	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
		longjmp	🟩	🟩	🟩	🟩	🟩*	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
jumping	stack	ret	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		baseptr	🟩*	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		funcptr	🟩	🟩*	🟩	🟩	🟩*	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		structfuncptr	🟩	🟩	🟩*	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
		longjmp	🟩*	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	data	funcptr	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		structfuncptr	🟩*	🟧	🟧	🟧	🟩	🟥	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
		longjmp	🟩	🟩	🟩	🟩	🟩	🟨*	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	bss	funcptr	🟩	🟩*	🟩	🟩	🟩	🟨	🟩*	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
		structfuncptr	🟩*	🟩	🟩	🟩	🟩*	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
		longjmp	🟩	🟩*	🟩	🟩	🟩	🟨	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

🟩=detected, 🟥=not detected + fp, 🟨=detected + fp, 🟧=not detected, *=manually validated results

follows. Thus, the pointer is incorrectly tainted. As the integration of the `add` into the `lea` would have required fewer bytes and would presumably execute faster, the compiler's reason for splitting them remains unclear.

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 stack objects are missing from the summary when optimizations are employed. A closer investigation again reveals incomplete debug information generated by the compiler to be at fault.

ASan detects the OOB write in 36% of all tests. Most failures can be attributed to the inability to detect intra-object OOB writes, as well as the reliance on red zones, preventing it from detecting jumping OOB writes. SoftBound successfully detects the OOB write in 73% of the test cases. All inter-object OOB writes are found, while intra-object OOB writes remain undetected.

5.3 Testing Real-world Programs

We evaluate Divak on three real-world vulnerabilities found in open source software. Besides, we run each program under test with a benign input to assess the false positives of Divak. To evaluate the performance of our static analysis, we assess the independent write and BNI matching performance. For the dynamic analysis, we collect statistics of three categories: (1) The number of dependent, independent, and unchecked writes; (2) Statistics about successful, ignored, and failed BNIs; (3) The successful pointee inferences from memory.

libxml (CVE-2017-9047). This vulnerability in the *libxml* library is a stack-based buffer overflow [4]. While attributed to the same root cause, OOB writes

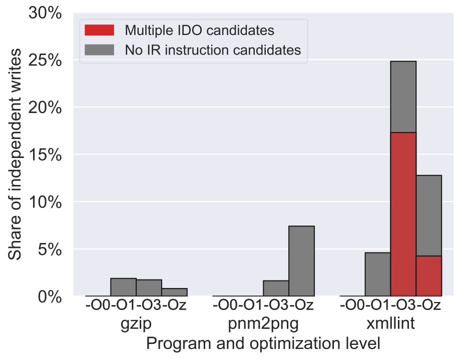

Fig. 4: Causes for unmatched independent writes.

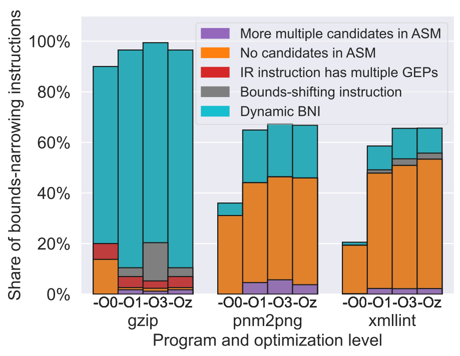

Fig. 5: Causes for unmatched BNIs.

Table 6: Detection performance for three real-world vulnerabilities.

	libxml (CVE-2017-9047)				libpng (CVE-2018-14550)				gzip (CVE-2001-1228)			
	0	1	3	z	0	1	3	z	0	1	3	z
DIVAK	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
ASan	▨	▨	▨	▨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
SoftBound	🟦	🟥	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

🟩=detected, 🟥=not detected + fp, ▨=partly detected, 🟦=not possible

occur at two different instructions. As shown in Table 6, DIVAK successfully detects both OOB writes at all four optimization levels and does not yield any false positives for the PoC or the benign input. Furthermore, due to its non-invasiveness, DIVAK is the only tool to detect all occurring OOB writes. As presented in Figure 4, independent write matching achieves a 0% failure rate without optimizations, which increases to 25% at `-O3` due to overlapping memory objects and missing line number debug information. BNI matching (Figure 5) also performs best without optimizations but has a higher failure rate, mostly due to a lack of line number debug information.

For this vulnerability, most checked writes are dependent. The share of checked writes ranges from 100% down to 31%, the latter being caused by one or more independent writes remaining unmatched. Most encountered BNIs concern pointers to unmonitored sections, mainly the heap. Last, determining a pointer's intended pointee object ranges from a 100% success rate at `-O0` to a 95% success rate at `-O3` for the PoC input. Most failures come from incomplete memory layout information caused by optimizations. Our assumption of memory object's lifetimes being representable by a single interval is responsible for at most 1% of failed pointee inferences. Manual validation of the affected objects identified by DIVAK yields that the results are largely correct and complete, except for a few objects that are not recorded in DWARF and thus not identified as affected.

While the OOB write affects many objects and stack frames at `-O0` and `-O1`, rearrangement of stack objects at `-O3` and `-Oz` causes it to only affect a single 5000-byte buffer. While the vulnerability can easily be used to divert control flow at `-O0` and `-O1` if no countermeasures are deployed, this is likely impossible at `-O3` and `-Oz` if the PoC cannot be modified to affect a larger range.

ASan detects the first OOB write at each optimization level. The second OOB write, however, is not detected, likely due to the overwritten byte not being located in a red zone. Instrumenting with SoftBound at `-O0` causes the compiler to crash. With employed optimizations, SoftBound reports an OOB write early during execution and crashes with a segmentation fault.

libpng (CVE-2018-14550) This stack-based buffer overflow is located in the *pnm2png* tool, part of the `libpng` library [21]. While attributed to the same root cause, OOB writes occur at two different instructions. As shown in Table 6, Divak successfully detects both OOB writes in pnm2png without raising false positives at any tested optimization level. For the benign input, however, false positives occur when optimizations are enabled. Manual validation of the affected objects identified by Divak reveals that the results are largely correct at all optimization levels, missing only an 8-byte object not present in DWARF when employing optimizations. Running an optimized version of pnm2png with a benign input using Divak causes multiple false positives in one function to be reported. Manually investigating the reason for this shows that the compiler optimized the zero-initialization of a struct by using a pointer to one of its fields for zeroing both the field and its sibling fields. ASan and SoftBound detect OOB writes at both locations for all tested optimization levels.

gzip (CVE-2001-1228) This is a `.bss` buffer overflow in the gzip utility. As shown in Table 6, Divak successfully detects the OOB write at all four optimization levels and yields no false positives for the PoC input. For the benign input, multiple false positives are reported. While independent write matching performs reasonably well (Figure 4), BNI matching works poorly (Figure 5). Only 10% are matched at `-O0` and less than 1% are matched at `-O3`. This is primarily caused by dynamic BNIs, which comprise between 70% and 85% of all BNIs. Furthermore, a non-negligible number of bounds-shifting instructions is found at `-O3`. Manual validation of Divak's output shows that all affected objects are correctly identified at all optimization levels. This demonstrates that the issues of an incomplete memory layout extraction are limited to the stack.

When using a benign input, three false positives are reported at `-O0` and `-O1`, and one is reported at `-O3` and `-Oz`. All of these are caused by our disregard for bounds-shifting instructions. In each case, a struct pointer passed to a function is used to access adjacent structs in the same array, causing Divak to incorrectly report OOB writes. Both ASan and SoftBound detect the vulnerability.

5.4 Performance Overhead

We evaluate Divak's overhead on the three real-world programs when feeding them benign inputs. We execute the programs using ASan, SoftBound, Divak, and natively under different optimization levels. We run each configuration ten times and consider the means. All measurements are performed on an Intel Xeon E3-1231v3 running Ubuntu 20.04. Table 7 shows the mean runtime overheads. Since our measurements indicate no substantial differences between optimization levels, we only present the results for `-O1`. As expected, Divak currently incurs a massive performance overhead (8,000 – 44,000×). ASan and SoftBound, on

Table 7: Performance overhead for each program at `-O1`.

Program	Divak	ASan	SoftBound
xmllint	8113×	5.4×	-
pnm2png	8592×	4.4×	3.9×
gzip	44097×	4.8×	1.5×

the other hand, incur at most a sixfold overhead. While Divak's performance overhead might seem to make it unusable in practice, it is important to note that little regard was given to performance during the implementation of this prototype, resulting in design decisions with a highly detrimental impact on performance. The most severe decision is the usage of S2E, which introduces a considerable overhead by executing most of the program's code in symbolic mode. We discuss a possible approach for reducing the overhead in Section 6.

6 Discussion

Our experiments show that Divak is well capable of characterizing OOB writes. While ASan and SoftBound cannot detect intra-object OOB writes, Divak's logic for detecting them is not perfect. 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. However, the former can be reduced by introducing a small number of heuristics, for example disregarding OOB writes relative to pointers created in the current function scope and not modified by pointer arithmetic. Despite these limitations, our experiments show that Divak's capabilities outperform ASan and SoftBound, with a false positive in the dependent write testbed being the only apparent downside.

Divak's main drawback is the excessive performance overhead. A promising approach to combat this is to replace the full-system emulation of S2E with dynamic binary instrumentation, e.g., Intel Pin. This would allow implementing pointer tracking by performing taint analysis through libdft [17], eliminating the overhead introduced by the use of symbolic variables. Although Pin would modify the program's memory layout by allocating space for its own metadata, the arrangement of objects within the program's sections would remain untouched. Thus, our goal of non-invasiveness would in practical terms be achieved. With the authors of libdft having measured an overhead of at most 6× for typical programs, our tool would experience a substantial decrease in overhead, even with a very conservative estimate of additional 50× overhead due to the amount of tainted data and our analysis logic. While Divak's need for a large number of taint colors would increase the load on libdft, limiting the set of colors by reusing them at the cost of a low chance for false negatives would be conceivable.

Thus, we conclude that, for use cases where only a low number of executions are necessary, e.g., bug triaging, a high recall is desirable, and false positives are tolerable, Divak is superior to instrumentation-based tools like ASan and SoftBound. This is especially true if intra-object OOB writes are to be detected. In addition to 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 faithfully characterize the *real* effects of vulnerabilities as they exist in programs deployed in production environments: DIVAK is the only approach that guarantees faithful results in terms of affected memory objects.

Limitations. DIVAK assumes that, once the bounds attached to a pointer are narrowed to a composite child, any derived pointer requires identical or narrower bounds. This does not hold for bounds-shifting instructions, causing false positives. Nevertheless, this can be mitigated by handling such instructions akin to BNIs. A drawback of DIVAK's reliance on DWARF debug information is a dependence on its correctness and completeness. While we did not encounter cases of incorrect information, we observed incomplete location descriptions under optimizations caused by compiler bugs. This occasionally causes DIVAK to fail tainting pointers or deliver incomplete memory layout results. However, as the analysis of such bugs gained traction in the past years [2,10], enabling them to be fixed, their impact on DIVAK's results can be expected to decrease.

Another current limitation is the assumption that a single interval can describe the lifetime of any object, as outlined in Section 4. This occasionally causes overlapping objects, which we try to combat by merging identically-sized objects. Nevertheless, this often leaves some overlapping objects in heavily inlined code, for which we observed up to 3% of objects to overlap with one another.

Lastly, we assume the program under test to be built without frame pointer omission and tail call optimizations, arguably violating DIVAK's non-invasiveness. Furthermore, we currently do not support position-independent executables. Both issues, however, are merely limitations of our current prototype that do not invalidate our results and can be alleviated with limited implementation effort.

Future work. A subject for future work is the extension of our design with omitted features, e.g., dynamic BNIs and bounds-shifting instructions. Moreover, re-implementing DIVAK to decrease its performance overhead is desirable to make it scalable as a part of other pipelines. Although DIVAK is meant to triage identified vulnerabilities and the discovery of new vulnerabilities is out of scope, a future research direction is the combination of DIVAK with approaches for finding alternative vulnerable paths, e.g., directed fuzzing, to create a more complete profile of the vulnerability capabilities. As the issue of invasiveness predominantly concerns the stack and globals, we do not consider heap-based OOB writes. However, DIVAK can be extended to intercept calls to memory allocators. Alternatively, one may use ASan's heap-based OOB analysis with disabled instrumentation of stack and global sections to largely maintain non-invasiveness.

7 Related Work

Several approaches have been proposed for detecting spatial memory bugs. Most of them rely on compile-time instrumentation (CTI) to insert their checking logic into the program [23,29,8,20], allowing for low overhead at the cost of highly invasive program modifications. Binary instrumentation-based approaches suffer from the lack of high-level semantic information, preventing them from providing strong spatial guarantees for the detection of certain OOB write types [33,31,37]. Similarly, binary-level pointer analysis [18] is often course-grained and cannot guarantee sufficient precision to track OOB writes that have marginal effects.

Identity-based approaches check whether the accessed memory locations are part of the expected object according to high-level program semantics. For accesses relative to pointers, this requires a sophisticated approach to maintain a mapping between pointers and intended pointee objects. *Pointer-based* approaches like Divak augment pointers with additional metadata, by embedding bounds information into pointers at pointer-creation sites [23,26,20,30,8]. *Object-based* approaches associate metadata only with memory objects, not with pointers [41], and test whether pointer arithmetic instructions have the same pointee before and after the operation. However, such approaches generally cannot detect intra-object OOB writes as composite objects overlap with their children.

Tripwire-based approaches [29,31] insert *red zones* around objects to detect OOB accesses. The main example of such approaches is ASan [29]. Its low-performance overhead makes it ideal for use cases such as fuzzing. A hardware-assisted variation for ARM64 [8] further decreases the memory overhead, while a kernel variation, KASAN [36] facilitates kernel fuzzing. One shortcoming is that non-contiguous OOB writes jumping over the red zone remain undetected. Furthermore, the insertion of new memory objects makes them invasive by design.

SoftBound [23] is a pointer-based sanitizer using CTI that provides relatively strong spatial detection guarantees but is heavily invasive and relies on wrappers for external function calls. While it promises to be able to detect intra-object OOB writes, we discovered this is not implemented in the publicly available tool.

Memcheck [31] and SGcheck [37] are tools for the Valgrind platform [24] and use a tripwire and heuristic-based approach, leveraging dynamic binary instrumentation without high-level semantic information. Orthogonally to our approach, QASan [13] detects heap memory violations. Intel MPX [26] is an AMD64 ISA extension that leverages CTI to get strong spatial guarantees. However, its deprecation caused its support to be removed from most compilers.

Besides in sanitizers, OOB write detection has application scenarios in larger pipelines. KOOBE [6] leverages KASAN and a pointer-based approach, similar to Divak, for heap-based kernel exploitation. BORG [25] discovers buffer overreads by employing a heuristic approach for recovering memory layout. Revery [38] employs a software-based memory tagging approach for heap-based AEG.

Finally, other related approaches identify memory errors at the LLVM IR level [35,34], however, they focus on different classes of bugs, such as memory leaks and use-after-free, which intrinsically require a less intrusive analysis.

8 Conclusion

We proposed Divak, a tool to detect OOB writes in a non-invasive manner and to distill their capabilities by identifying the affected source code-level objects stored in memory. Using two benchmarks and three real-world vulnerabilities, we showed that Divak can keep up with, and in some cases even exceed, the detection performance of current instrumentation-based OOB write detection approaches, yielding negligible false positives, at the cost of higher overhead.

References

1. Anderson, J.P.: Computer Security Technology Planning Study. Tech. rep., U.S. Air Force Electronic Systems Division (1972)
2. Assaiante, C., D'Elia, D.C., Di Luna, G.A., Querzoni, L.: Where Did My Variable Go? Poking Holes in Incomplete Debug Information. In: Procs. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2023)
3. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: automatic exploit generation. In: Procs. of the Network and Distributed System Security Symposium (NDSS) (2011)
4. Böhme, M.: oss-security - Invalid writes and reads in libxml2 (2017)
5. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on Binary Code. In: Procs. of the IEEE Symposium on Security and Privacy (S&P) (2012)
6. Chen, W., Zou, X., Li, G., Qian, Z.: KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In: Procs. of the USENIX Security Symposium (2020)
7. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In: Procs. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2011)
8. Clang: Hardware-Assisted AddressSanitizer Design Documentation (2022)
9. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Procs. of the USENIX Security Symposium (1998)
10. Di Luna, G.A., Italiano, D., Massarelli, L., Österlund, S., Giuffrida, C., Querzoni, L.: Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries. In: Procs. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2021)
11. Ding, Z.Y., Goues, C.L.: An Empirical Study of OSS-Fuzz Bugs (2021)
12. Donovan, A.A., Kernighan, B.W.: The Go Programming Language. Addison-Wesley Professional (2015)
13. Fioraldi, A., D'Elia, D.C., Querzoni, L.: Fuzzing binaries for memory safety errors with qasan. In: Procs. of the IEEE Secure Development Conference (2020)
14. Heelan, S.: Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. Master's thesis, University of Oxford (2009)
15. Huang, S.K., Huang, M.H., Huang, P.Y., Lai, C.W., Lu, H.L., Leong, W.M.: CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In: Procs. of the IEEE International Conference on Software Security and Reliability (SERE) (2012)
16. ISO Central Secretary: Programming languages — C. Standard ISO/IEC 9899:2011, International Organization for Standardization, Geneva, CH (2011)
17. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In: Procs. of the 8th ACM Conference on Virtual Execution Environments (2012)
18. Kim, S.H., Zeng, D., Sun, C., Tan, G.: Binpointer: towards precise, sound, and scalable binary-level pointer analysis. In: Procs. of the ACM International Conference on Compiler Construction (2022)
19. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press (2018)
20. Kroes, T., Koning, K., Kouwe, E.v.d., Bos, H., Giuffrida, C.: Delta Pointers: Buffer Overflow Checks Without the Checks. In: Procs. of the EuroSys Conference (2018)

21. Luo, Z.: Stack-buffer-overflow in pnm2png in function get_token (2018)
22. MITRE Corporation: CWE Top 25 Most Dangerous Software Weaknesses (2021)
23. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In: Procs. of the ACM Conference on Programming Language Design and Implementation (PLDI) (2009)
24. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Procs. of the ACM Conference on Programming Language Design and Implementation (PLDI) (2007)
25. Neugschwandtner, M., Comparetti, P.M., Haller, I., Bos, H.: The BORG: Nanoprobing Binaries for Buffer Overreads. In: Procs. of the ACM Conference on Data and Application Security and Privacy (CODASPY) (2015)
26. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., Fetzer, C.: Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches (2017)
27. PaX Team: Address Space Layout Randomization (2001)
28. Rosier, H.: ripe64 (2019), https://github.com/hrosier/ripe64
29. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker. In: Procs. of the USENIX Annual Technical Conference (2012)
30. Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyrklevich, V., Vyukov, D.: Memory Tagging and how it improves C/C++ memory safety (2018)
31. Seward, J., Nethercote, N.: Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In: Procs. of the USENIX Annual Technical Conference (2005)
32. Shoshitaishvili, Y., Weissbacher, M., Dresel, L., Salls, C., Wang, R., Kruegel, C., Vigna, G.: Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In: Procs. of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (2017)
33. Slowinska, A., Stancescu, T., Bos, H.: Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In: Procs. of the USENIX Annual Technical Conference (2012)
34. Sui, Y., Xue, J.: Svf: interprocedural static value-flow analysis in llvm. In: Procs. of the ACM International Conference on Compiler Construction (2016)
35. Sui, Y., Ye, D., Xue, J.: Static memory leak detection using full-sparse value-flow analysis. In: Procs. of the International Symposium on Software Testing and Analysis (2012)
36. The kernel development community: The Kernel Address Sanitizer (KASAN) — The Linux Kernel documentation (2021)
37. Valgrind Developers: SGCheck: An Experimental Stack and Global Array Overrun Detector (2012), https://valgrind.org/docs/manual/sg-manual.html
38. Wang, Y., Zhang, C., Xiang, X., Zhao, Z., Li, W., Gong, X., Liu, B., Chen, K., Zou, W.: Revery: From Proof-of-Concept to Exploitable. In: Procs. of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (2018)
39. Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., Joosen, W.: RIPE: Runtime Intrusion Prevention Evaluator. In: Procs. of the Annual Computer Security Applications Conference (ACSAC) (2011)
40. Xu, L., Jia, W., Dong, W., Li, Y.: Automatic Exploit Generation for Buffer Overflow Vulnerabilities. In: Procs. of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS) (2018)
41. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In: Procs. of the ACM Symposium on Information, Computer and Communications Security, (ASIACCS) (2010)