# Locating Vulnerabilities in Binaries via Memory Layout Recovering

Haijun Wang
Shenzhen University
China

Xiaofei Xie
Nanyang Technological University
Singapore

Shang-Wei Lin
Nanyang Technological University
Singapore

Yun Lin
National University of Singapore
Singapore

Yuekang Li
Nanyang Technological University
Singapore

Shengchao Qin
Teesside University
United Kingdom

Yang Liu
Nanyang Technological University
Singapore

Ting Liu
Xi'an Jiaotong University
China

## ABSTRACT

Locating vulnerabilities is an important task for security auditing, exploit writing, and code hardening. However, it is challenging to locate vulnerabilities in binary code, because most program semantics (e.g., boundaries of an array) is missing after compilation. Without program semantics, it is difficult to determine whether a memory access exceeds its valid boundaries in binary code. In this work, we propose an approach to locate vulnerabilities based on memory layout recovery. First, we collect a set of passed executions and one failed execution. Then, for passed and failed executions, we restore their program semantics by recovering fine-grained memory layouts based on the memory addressing model. With the memory layouts recovered in passed executions as reference, we can locate vulnerabilities in failed execution by memory layout identification and comparison. Our experiments show that the proposed approach is effective to locate vulnerabilities on 24 out of 25 DARPA's CGC programs (96%), and can effectively classifies 453 program crashes (in 5 Linux programs) into 19 groups based on their root causes.

## 1 INTRODUCTION

For memory unsafe languages like C/C++, memory corruption vulnerability is one of the most severe defects, as it can lead to software crash or even allows adversaries to take full control of the software. Buffer overflow is one of the most common memory corruption vulnerabilities, which is also the focus of this paper. In the remaining of this paper, by vulnerability, we mean buffer overflow vulnerability.

There have been a number of techniques [1–4] that can locate buffer overflows, and most of them are in the source code level, such as AddressSanitizer [1]. However, the source code is not always available (e.g., closed-source software and off-the-shelf components in IoT devices). Further, it is shown that the semantics of binary code may be different from its source code [5]. For the purpose of binary security auditing, exploit writing and code hardening, it is highly significant to locate buffer overflows in binary code directly.

However, it is much more challenging to locate buffer overflows in binaries than in source code. When the source code is compiled into binaries, its program semantics is missing, i.e., we are not able to identify variables of program and their memory boundaries anymore. Without the memory boundaries, locating buffer overflows in binary code becomes very difficult. Although there have been some techniques working on the binary code (e.g., Valgrind Memchecks [3]), none of them can locate buffer overflows within the stack/global memory regions (e.g., overflow beyond an array but still within its resident stack frame) [1]. To address these issues, recovering the program semantics, i.e., memory boundaries of variables, is necessary to locate buffer overflows in binary code.

In addition, locating vulnerabilities highly benefits triaging program crashes in binaries [6]. It is well-known that the same vulnerability can produce various symptoms, leading to crashes at different locations. For example, the fuzzing system (e.g., fairfuzz, aflfast and aflgo [7–11]) usually generates a large number of crashes. However, not all of these crashes are unique. Many of them are due to the same vulnerability. If the crashes can be grouped according to their root causes, it would greatly improve the efficiency of analysis.

Aiming at addressing the above challenges, we propose an approach, based on dynamic execution information, to locate buffer overflows in binaries. Our approach mainly consists of two parts: *recovering memory layout* and *locating vulnerabilities*.

**Recovering Memory Layout**. A memory layout represents the static data structure of a variable in source code, e.g., a data structure with its members. In our approach, we actually use the dynamic execution information in binary code to restore static data structure information of variables in source code. To recover the memory layout, first, we identify the relevant addressing instructions for each memory access in dynamic execution information. Second, we recover a memory layout for each memory access based on the memory addressing model. Third, if multiple memory layouts (e.g., recovered from multiple executions) access the same variable, we merge them into a memory layout, generating a more complete static data structure for that variable. Compared to existing techniques [12–16], our approach can precisely recover fine-grained memory layouts of variables (c.f. Section 6.1).

**Locating Vulnerabilities**. Based on recovered memory layouts, we locate buffer overflows in a failed execution. To achieve this goal, we need to determine whether the recovered memory layout in failed execution exceeds its valid boundaries. To infer the boundaries, we collect a failed execution and a set of passed executions. In this paper, the failed execution means to cause the program crash,

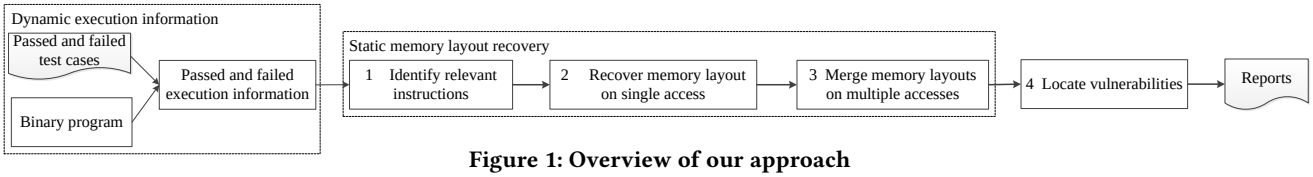
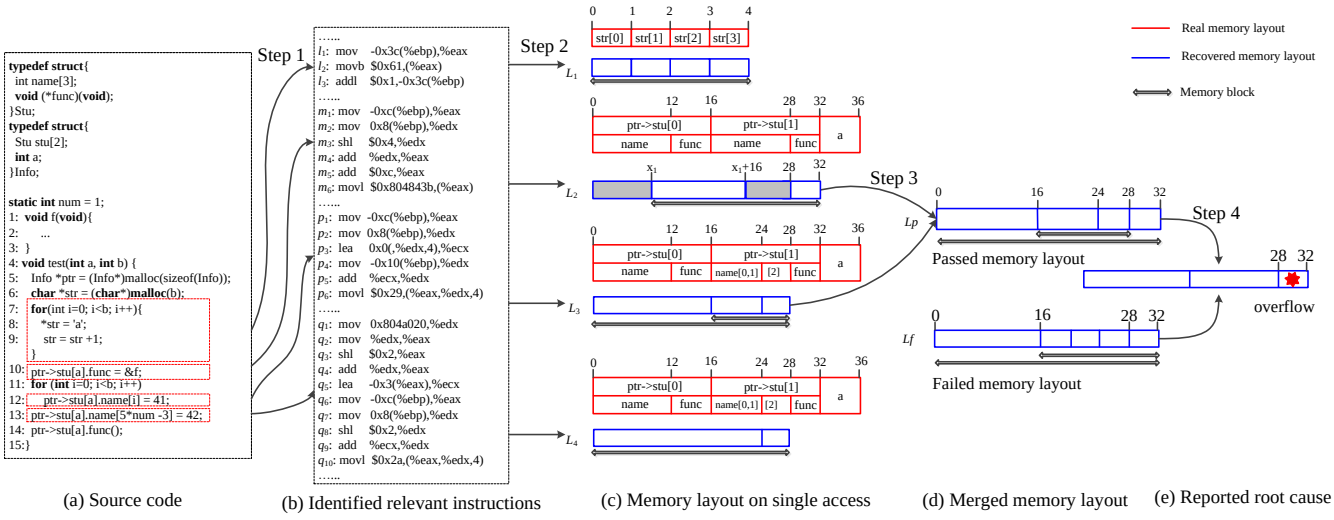



**Figure 1: Overview of our approach**




(a) Source code (b) Identified relevant instructions (c) Memory layout on single access (d) Merged memory layout (e) Reported root cause

**Figure 2: Illustration example for our approach**

while the passed execution would not. The failed execution is the input to our approach, and passed executions can be obtained easily from existing tools or test suites (c.f. Section 5.2). For passed executions, we recover their memory layouts (called passed memory layouts), which are considered as the memory layouts within valid boundaries. For the failed execution, we also recover its memory layouts (called failed memory layouts), which may include memory layouts exceeding valid boundaries (called vulnerable memory layouts). With the memory layouts recovered in passed executions as reference, we can locate vulnerable memory layout, i.e., candidate buffer overflow, in the failed execution by memory layout identification and comparison (see Section 4).

We have implemented the proposed approach in a prototype tool, and evaluated its effectiveness in two different aspects: 1) locating vulnerabilities, and 2) triaging program crashes. In locating vulnerabilities, our approach is effective to locate buffer overflow vulnerabilities on 24 out of 25 DARPA's CGC programs (96%). In triaging experiments, our approach is able to classify the 453 program crashes (in 5 widely used Linux programs) into 19 groups, while AFL reports 320 unique crashes (groups).

The contributions of this work are summarized as follows:

- We formalize a memory addressing model, based on which, together with the dynamic execution information, we propose a general approach to precisely recover the static fine-grained hierarchical memory layouts of program variables.
- With the memory layouts recovered in passed executions as reference, we propose an approach to locate buffer overflows in failed execution by memory layout identification and comparison. To the best of our knowledge, by using fine-grained memory layouts, our approach is the first work to locate buffer overflows within stack, global memory regions, and data structures in binary code.
- We implemented a prototype of the proposed approach and evaluated its effectiveness on binary programs with diverse kinds of vulnerabilities.

## 2 APPROACH OVERVIEW

### 2.1 Motivating Example

Fig. 2 (a) shows a program with a buffer overflow vulnerability. If the input of function *test* is (1,4), the program crashes at Line 14. The buffer overflow is triggered in the loop (Line 11). The variable $ptr \rightarrow stu[1].name$ contains three elements, but is assigned with four elements. As a result, the variable $ptr \rightarrow stu[1].func$ is overwritten, and the program crashes when Line 14 is executed (where $ptr \rightarrow stu[1].func$ is accessed).

After being compiled into binaries, the program semantics is missing, making it difficult to identify the variables and their boundaries. For example, the variable $ptr \rightarrow stu[1].name$ and its memory boundaries (its size is 12 bytes) are missing in binaries. Without this information, the buffer overflow cannot be identified when it is written with 16 bytes.

### 2.2 Overview of our Approach

To address this issue, we propose an approach to locate buffer overflows in binaries. The overall flow of the proposed approach is shown in Fig. 1. The input to our approach is a set of concrete[1] execution information (either passed or failed).

[1]Notice that we call the obtained executions "concrete" because they are real traces from dynamic executions. Thus, there are no loop or recursive function issues any more in our approach as they have been unrolled during dynamic executions.

The proposed approach consists of four steps. In the first step, given passed and failed executions, we identify relevant addressing instructions for each memory access. In the second step, we recover the memory layout for each memory access. Since one execution may have multiple memory accesses, we may have more than one memory layout recovered. If some of the recovered memory layouts access the same variable, we merge them into one in the third step. After this step, we have recovered memory layouts for each execution (either passed or failed). In the fourth step, we use the memory layouts of passed executions as reference to locate buffer overflows in the failed execution. In the following, we walk through these steps using the motivating example in Fig. 2.

**Step** 1. We identify relevant addressing instructions for each memory access. To access a variable, the program first determines its memory address. For a memory access instruction, we perform a backward taint analysis to identify the relevant instructions used to compute its memory address. Fig. 2 (b) shows several sets of identified instructions. For example, $m_6$ is a memory access instruction, we perform backward taint analysis and identify the relevant instructions $m_1 \ldots m_6$, which correspond to Line 10 in Fig. 2 (a).

**Step** 2. We recover memory layout based on identified instructions for each memory access. In general, the memory address of a variable is computed by iteratively adding an offset to the address of its enclosing variable. Based on this iterative process, we identify the *memory blocks* (c.f. Definition 1), which represent the enclosed variables. In addition, we construct their hierarchical structure, and finally form a memory layout.

For example, consider instructions $p_1 \ldots p_6$, which correspond to Line 12 (e.g., $ptr \rightarrow stu[1].name[2] = 41$). The recovered memory layout is $L_3$, as shown in Fig. 2 (c), where the red part is the real memory layout of the variable, and the blue part is the recovered memory layout (double-headed arrows below indicate memory blocks). Based on identified instructions, we can infer that: *the memory address is computed by adding two offsets (16 and 8) to the base address, and then is written with four bytes.* Based on this information, we identify two memory blocks [0, 28] and [16, 28], where 0 represents a relative base address. In addition, we construct their hierarchical structures. The memory block [16, 28] is enclosed in [0, 28]. The identified memory blocks actually correspond to the variables in source code. For example, [0, 28] and [16, 28] represents arrays $ptr \rightarrow stu$ and $ptr \rightarrow stu[1].name$, respectively. The hierarchical structures reflect the enclosing/enclosed relationships of variables. The variable $ptr \rightarrow stu[1].name$ is enclosed in $ptr \rightarrow stu$.

**Step** 3. When multiple memory accesses operate on the same variable in the executions (e.g., in the unrolled loop), corresponding recovered memory layouts should be merged. For example, the memory layouts $L_2$ and $L_3$ in Fig. 2 (c) represent the same variable, we merge them into a more complete memory layout $L_p$, as shown in Fig. 2 (d), which reflects not only the variable $ptr \rightarrow stu[1].func$ but also $ptr \rightarrow stu[1].name$.

**Step** 4. With above three steps, we recover memory layouts of variables. To locate vulnerabilities, we need to determine whether the recovered memory layout in failed execution exceeds its valid boundaries. With the memory layouts recovered in passed executions as reference, we locate buffer overflows in failed execution by memory layout identification and comparison. For example, we recover a passed memory layout , i.e., *test*(1,3), and a failed memory layout , i.e., *test*(1,4), as shown in $L_p$ and $L_f$ of Fig. 2 (d). By memory layout identification, memory blocks [16, 28] in $L_p$ and [16, 32] in $L_f$ are compared, and [28, 32] (i.e., $ptr \rightarrow stu[1].func$) in $L_f$ is considered to be overflowed, as shown in Fig. 2 (e).

The proposed approach can help users in two scenarios: (1) One wants to disassemble or debug the binary program. With the recovered fine-grained memory layouts as debug symbols, he/she can interpret some key data structures. (2) One has a binary program crashed, and he/she wants to figure out whether the crash is due to buffer overflow and its root causes. Given the failed (crashed) execution, together with a set of passed executions, our approach can diagnose whether there is a buffer overflow and its root causes.

# 3 MEMORY LAYOUT RECOVERY

We first formalize a memory addressing model, and then introduce the memory layout recovery based on memory addressing model.

## 3.1 Memory Addressing Model

Before a variable is accessed, its memory address needs to be determined first. In binary code, there are usually two addressing modes for memory access: direct and indirect addressing [12]. In direct addressing, the address is encoded in the instruction itself, usually used to access a scalar variable.

The indirect addressing mode is typically used to access an array or a data structure. Generally, the address is computed by an equation: $address = base + (index * scale) + offset$, where *index* represents the index of an array, *scale* is the size of unit element in the array, and *offset* implies the offset calculation for the member of a data structure.

The equation for indirect addressing depends on the hierarchical structure of a variable. For example, a data structure may contain an array as its member or the element of an array can be a data structure. Hence, there could be more than one $index * scale$ and *offset*. A more general equation is:

$$address = base + \sum_{i=1}^{n}(index_i * scale_i) + offset \quad (1)$$

In general, *offset* may be an optimized value due to the compilation, i.e., $offset = \sum_{i=1}^{n} offset_i$, where each $offset_i$ corresponds to one $index_i * scale_i$ and $offset_i \geq 0$. The addressing equation is the key insight for recovering memory layouts of variables in this paper.

Example 1. Consider $ptr \rightarrow stu[index_1].name[index_2]$ in Fig. 2(a), the memory address is calculated by: $base + offset_1 + index_1 * scale_1 + offset_2 + index_2 * scale_2$, where $base = ptr$ is the base address; $offset_1 = 0$ is the offset of *stu* in data structure *ptr*; $index_1 * scale_1$ calculates the address of the $index_1{}^{th}$ element of *stu*; $offset_2 = 0$ is the offset of *name* in data structure $stu[index_1]$; $index_2 * scale_2$ calculates the address of the $index_2{}^{th}$ element of *name*.

## 3.2 Definition of Memory Layout

Definition 1. *A memory block m is a tuple* $(m^{\llcorner}, m^{\lrcorner}, \widetilde{m})$*, where* $m^{\llcorner}$ *is the start address and* $m^{\lrcorner}$ *is the end address. If it represents an array,* $\widetilde{m}$ *is the size of its unit element; Otherwise,* $\widetilde{m}$ *is zero. We use* $\overleftrightarrow{m}$ *to denote the size of m.*

Definition 2. *A memory layout* $L = (m, M, E)$ *is a directed acyclic graph (DAG), where m is the root memory block, M is a set of*

*memory blocks, and $E \subset M \times M$ is a set of directed edges connecting memory blocks such that $(m_1, m_2) \in E$ if $m_1^{\llcorner} \leq m_2^{\llcorner} \wedge m_2^{\lrcorner} \leq m_1^{\lrcorner}$.*

Definition 3. *Given two memory blocks $m_1$ and $m_2$, if $m_1$ and $m_2$ represent the same variable, they are the alias memory blocks; if the variable represented by $m_1$ is enclosed in that represented by $m_2$, then $m_1$ is an inner memory block of $m_2$.*

Specifically, a directed edge $(m_1, m_2)$ in memory layout $L$ represents that $m_2$ is the inner memory block of $m_1$. A memory layout actually reflects the static hierarchical data structure of a variable (i.e., the enclosing/enclosed relationship).

Theorem 1. *Given two memory blocks $m_1$ and $m_2$, if they are intersected (i.e., $m_1^{\llcorner} < m_2^{\lrcorner} \wedge m_1^{\lrcorner} > m_2^{\llcorner}$), and $0 < \widetilde{m_2} \leq \widetilde{m_1}$ is true, $m_2$ is the alias or inner memory block of $m_1$.*

Theorem 2. *Given two memory blocks $m_1$ and $m_2$, if they are intersected (i.e., $m_1^{\llcorner} < m_2^{\lrcorner} \wedge m_1^{\lrcorner} > m_2^{\llcorner}$), and $0 < \widetilde{m_2} \leq \widetilde{m_1} \wedge \overleftarrow{m_2} > \widetilde{m_1}$ is true, $m_2$ is the alias memory block of $m_1$.*

Theorems 1 and 2 present two basic approaches to determine the relationship between two memory blocks. Their detailed proofs can be found in the website [17].

## 3.3 Memory Layout Recovery on Single Memory Access

Memory layouts are recovered based on memory addressing model. For the direct addressing, the recovered memory layout contains only one memory block $m$, i.e., the accessed memory block, and its unit element size is zero (i.e., $\widetilde{m} = 0$). For the indirect addressing, the memory layout is recovered based on the following four steps:

*3.3.1 Identifying Relevant Instructions.* We first identify the relevant instructions, which compute the address for the memory access. To achieve this goal, we perform a backward taint analysis, in which the memory access instruction is regarded as the *sink*. Different from traditional taint analysis, we only propagate the taints among the registers (not including registers `esp` and `ebp`), since the memory address is computed by registers [18]. For example, in Fig. 2(b) Line $m_6$ is a memory access instruction. Based on the taint propagation among registers, we continue to identify Lines $m_1 - m_5$. At Lines $m_1$ and $m_2$, we stop the taint propagation as their source operands are memory, not registers. As a result, the identified addressing instructions are Lines $m_1$-$m_6$.

*3.3.2 Recovering Addressing Equation.* After identifying relevant instructions, we recover the addressing equation, based on the address calculation in identified instructions, in the form of Equation 1. For example, we identify instructions (Lines $m_1$-$m_6$) in Fig. 2(b), where Line $m_6$ is a memory access instruction. At Line $m_1$, it stores the address pointed by *ptr* to register *eax*, which is the base address. At Line $m_2$, it stores the value of variable *a* to register *edx*, which is $index_1$. Then, $index_1$ is multiplied by 16 at Line $m_3$, and thus $scale_1$ is 16. At Line $m_4$, it adds $index_1 * scale_1$ to the base address. At Line $m_5$, it adds *offset* (i.e., 0*xc*) to compute the memory address. Thus, the recovered addressing equation is: $eax + (edx * 16) + 12$.

*3.3.3 Optimizing Addressing Equation.* The addressing equation is optimized as follows: 1) Sorting $n$ terms $index_i * scale_i$ based on the descending order of $scale_i$ for $i \in \{1, \ldots, n\}$. 2) Merging some terms of $index_i * scale_i$ if possible. For two adjacent terms $index_i * scale_i$ and $index_{i+1} * scale_{i+1}$ $(1 \leq i < n)$, if $scale_i < index_{i+1} * scale_{i+1}$, they are merged into one term $index * scale$, where $scale = \gcd(scale_i, scale_{i+1})$ (i.e., greatest common divisor) and $index = (index_i * scale_i + index_{i+1} * scale_{i+1})/scale$. In fact, these two terms are used together to access the same array. The detailed explanation can be found in the website [17]. For example, the equation for array access *int a[2p + q]* is recovered as: *base* $+p * (2 * sizeof(int)) + q * sizeof(int)$, where $p * (2 * sizeof(int))$ and $q * sizeof(int)$ are used together to access an array. Hence, they should be merged.

**Algorithm 1:** `GenerateLayout`

**input** : $address = base + \sum_{i=1}^{n}(index_i * scale_i) + offset$
**output**: a memory layout $L = (m, M, E)$

```
1  Let offset = Σ_i^n x_i, where x_i represents offset_i and x_i ≥ 0;
2  start ⟵ base;
3  end ⟵ address + sizeof(accessed memory) ;
4  if ∀1 ≤ i ≤ n: index_i*scale_i ≥ 0 then
5      M ⟵ ∅, E ⟵ ∅ ;
6      for i = 1 : n do
7          start ⟵ start + x_i;
8          m_i ⟵ (start, end, scale_i);
9          M ⟵ M ∪ {m_i} ;
10         if i ≥ 2 then
11             E ⟵ E ∪ (m_{i-1}, m_i) ;
12         start ⟵ start + index_i*scale_i ;
13     m ⟵ m_1 ;
14 else
15     m ⟵ (start, end, 0) ;
16     M ⟵ {m}, E ⟵ ∅ ;
17 return L ;
```

*3.3.4 Recovering Memory Layout.* Based on the optimized addressing equation, we recover the memory layout in Algorithm 1. It takes the addressing equation as input, and outputs a recovered memory layout. As described in Section 3.1, *offset* in the equation may be an optimized value. Thus, at Line 1 we introduce the parameter $x_i$ to represent each possible $offset_i$. We recover the memory layout based on two cases: (1) if every $index_i * scale_i$ in the equation is greater than or equal to zero, the memory layout can be recovered normally (Lines 4–13). We iteratively identify the memory blocks and construct their hierarchical structure (Lines 6–12). In this loop, we first compute the start address of a memory block by adding the offset $x_i$ (Line 7). Then, we recover this memory block $m_i$ at Line 8, and its unit size is $scale_i$. At Line 11, we add a directed edge $(m_{i-1}, m_i)$. Last, we calculate the address of the $index_i{}^{th}$ element and continue to recover the next memory block (Line 12). (2) If there is an $index_i * scale_i$ that is negative (i.e., $index_i$ is negative), it may be used to access an array with other $index_j * scale_j$ together. For example, in $ptr \rightarrow stu[a].name[5 * num - 3]$, -3 (i.e., $index_i$) is used to access array *name* with $5 * num$ (i.e., $index_j$) together. However, we cannot determine whether *a* or $5 * num$ is used to access an array together. In this case, we adopt a conservative strategy to only recover the largest memory block (Line 15).

Example 2. In Fig. 2(a), we assume that the address pointed by pointer *ptr* is 0 and the value of variable *a* is 1. Based on Lines $m_1-m_6$ in Fig. 2(b), we can recover an equation: $0+1*16+12$. We use $x_1$ and $x_2$ to represent the optimized offsets, i.e., $x_1+x_2=12$. Based on Line 8 of Algorithm 1, a memory block $(x_1, 32, 16)$ is identified. Hence, we recover a memory layout, as $L_2$ in Fig. 2(c). For Lines $q_1-q_{10}$, we recover an equation: $0+1*20+1*16+(-3*4)$. Since $index_3 * scale_3$ (i.e., $-3*4$) is negative, we only recover the largest memory block $(0, 28, 0)$, as $L_4$ in Fig. 2(c).

## 3.4 Memory Layout Recovery on Multiple Memory Accesses

When multiple memory accesses operate on the same variable (e.g., in the unrolled loop), corresponding recovered memory layouts should be merged to generate a more complete one. Notice that multiple memory accesses could happen in the same or different executions. It is non-trivial to infer which memory layouts can be merged because their concrete memory addresses cannot be used as a unique identification. Hence, we first index the memory layouts to make their memory addresses in a relative coordinate system so that we can determine whether they can be merged.

*3.4.1 Indexing Memory Layout.* Assume that the address space of a program consists of several non-overlapping memory-regions [12, 19]: stack, heap, and global, which correspond to functions, heap-allocation statements and global/static variables, respectively. Specifically, each function has a memory-region, i.e., its stack frame; one heap-allocation statement has a memory-region; the data section is a memory-region including global/static variables. Hence, the concrete memory address can be indexed by a pair: (memory-region, offset). The indexing process is described as follows:

- For variable *a* which is a local variable in a function $f$ (stack memory), we index its memory address by the pair $(f, \&a - fp)$, where $f$ represents the memory region associated with $f$, $\&a$ is the concrete memory address of *a*, and *fp* is the frame pointer of $f$ (e.g., register ebp).
- For variable *a* which is an enclosed variable (e.g., the member of a data structure) in heap memory allocated at statement *s*, we index its memory address by the pair $(s, \&a - ptr(s))$, where *s* represents the memory region associated with the heap-allocation statement *s*, $\&a$ is the concrete memory address of *a*, and $ptr(s)$ is the base address of memory allocated at statement *s*.
- For variable *a* which is a global/static variable in global memory, we index its memory address by the pair $(g, \&a - ds)$, where $g$ is the memory region associated with data section (often denoted *.data* in binary code), $\&a$ is the concrete memory address of *a*, and *ds* is the base address of data section.

After indexing, the addresses of memory layouts are in the same coordinate, i.e., they are relative to the beginning of a memory-region. Hence, we can merge the memory layouts.

*3.4.2 Merging Memory Layouts.* Given two memory layouts $L_1$ and $L_2$, we merge them based on two cases, as shown in Algorithm 2. The first case is that their root memory blocks $m_1$ and $m_2$ are intersected (Line 1). We first merge the root memory blocks of $L_1$ and $L_2$ by `MergeBlock` (i.e., Algorithm 3), and construct a new memory layout $L'$. Then, we continue to merge the children of $L'$ by `UpdateLayout` (i.e., Algorithm 5). The second case is that $m_1$ and $m_2$ are not intersected. They represent different variables, and cannot be merged. Next, we introduce the merging in the first case.

**Algorithm 2:** `MergeLayout`

**input** : memory layouts $L_1$=($m_1$,$M_1$,$E_1$) and $L_2$=($m_2$,$M_2$,$E_2$)
**output**: merged memory layout $L'$=($m'$, $M'$, $E'$)

```
1 if m1^⌞ < m2^⌟ ∧ m1^⌟ > m2^⌞ then
2     L' ⟵ MergeBlock(L1, L2);
3     L' ⟵ UpdateLayout(L') ;
4     return L' ;
5 else return NIL ;
```

**Algorithm 3:** `MergeBlock`

**input** : memory layouts $L_1$=($m_1$,$M_1$,$E_1$) and $L_2$=($m_2$,$M_2$,$E_2$)
**output**: merged memory layout $L'$=($m'$, $M'$, $E'$)

```
1  if 0 < m̃1 < m̃2 then swap(L1, L2) ;
2  (m', M', E') ⟵ (m1, M1 ∪ M2, E1 ∪ E2);
3  (m'^⌞, m'^⌟, m̃') ⟵ (min(m1^⌞, m2^⌞), max(m1^⌟, m2^⌟), m̃1);
4  status ⟵ DetermineLevel(m', m2);
5  if m2 is the inner memory block of m' in status then
6      E' ⟵ E' ∪ {(m', m2)};
7  else
8      foreach mc ∈ Child(m2) do
9          E ⟵ E \ {(m2, mc)} ;
10         E ⟵ E ∪ {(m', mc)} ;
11     M ⟵ M \ {m2};
12     if m̃2 == 0 then m̃' ⟵ 0 ;
13 return L' ;
```

Algorithm 3 introduces how to merge root memory blocks $m_1$ and $m_2$ of $L_1$ and $L_2$. At Line 1, it makes sure that $\widetilde{m_2} \leq \widetilde{m_1}$ is true. Thus, $m_2$ is the alias or inner memory block of $m_1$ (c.f. Theorem 1). At Lines 2-3, it merges $m_1$ and $m_2$ as a new memory block $m'$, and constructs a new memory layout $L'$. Then, it determines whether $m_2$ is the inner memory block of $m'$ by `DetermineLevel` (i.e., Algorithm 4). If so, $m_2$ is added as the inner memory block of $m'$ (Line 6). Otherwise, $m_2$ and $m'$ are alias memory blocks, or their relationship cannot be determined. It adds the inner memory blocks (i.e., children) of $m_2$ as the inner memory blocks of $m'$ (Lines 8-10), and deletes $m_2$ (Line 11). Note that, when the relationship between $m_2$ and $m'$ cannot be determined, the algorithm ignores the case that $m_2$ is the inner memory block of $m'$. As a result, it may lose some precision but is still correct. If the unit size of $m_2$ is zero, the unit size of $m'$ is updated as zero (Line 12).

Given two memory blocks $m_1$ and $m_2$ such that $\widetilde{m_2} \leq \widetilde{m_1}$ (c.f. Algorithm 3), Algorithm 4 determines whether $m_2$ is the alias or inner memory block of $m_1$. At Lines 1-3, it decides that $m_2$ is the alias memory block of $m_1$ (c.f. Theorem 2). If it cannot determine $m_2$ is the alias memory block of $m_1$, we consider the inner memory block $m_c$ of $m_1$ at Lines 5-11. It first checks whether $m_2$ and $m_c$ are intersected (Line 6). Then, it checks whether $\widetilde{m_2} \leq \widetilde{m_c}$ is true (Line 7). If so, $m_2$ is the alias or inner memory layout of $m_c$ (c.f. Theorem 1). Based on transitivity, $m_2$ is the inner block of $m_1$. At Line 10, it checks whether $m_2$ is the alias block of $m_c$ (c.f. Theorem 2). If so, $m_2$ is also the inner block of $m_1$ based on transitivity.

**Algorithm 4:** `DetermineLevel`

**input** : memory blocks $m_1$ and $m_2$
**output**: determining relationship between $m_1$ and $m_2$

```
1  if m̃1 ≠ 0 and m̃2 ≠ 0 then
2      if ⟷m2 > m̃1 then
3          return m2 is alias memory block of m1;
4      else
5          foreach mc ∈ Child(m1) do
6              if mc^⌞ < m2^⌟ and mc^⌟ > m2^⌞ then
7                  if m̃2 ≤ m̃c then
8                      return m2 is inner memory block of m1;
9                  else
10                     if ⟷mc > m̃2 then
11                         return m2 is inner block of m1;
12 return it is unknown
```

**Algorithm 5:** `UpdateLayout`

**input** : memory layout $L = (m, M, E)$
**output**: updated memory layout $L' = (m, M', E')$

```
1  L' ⟵ L ;
2  Let Q be an empty queue ;
3  Q.ENQUEUE(m) ;
4  while Q is not empty do
5      mq ⟵ Q.DEQUEUE() ;
6      foreach (m1, m2) s.t. m1, m2 ∈ Child(mq) do
7          Let L1 = (m1, M1, E1) and L2 = (m2, M2, E2) be two sub
           layouts of L', whose roots are m1 and m2;
8          if m1^⌞ < m2^⌟ and m1^⌟ > m2^⌞ then
9              L3 : (m3, M3, E3) ⟵ MergeBlock(L1, L2) ;
10             M' ⟵ M' \ (M1 ∪ M2) ∪ M3 ;
11             E' ⟵ E' \ (E1 ∪ E2) ∪ E3 ;
12             E' ⟵ E' \ {(mq, m1), (mq, m2)}∪{(mq, m3)} ;
13             L' ⟵ UpdateLayout(L');
14             return L';
15     foreach mc ∈ Child(mq) do
16         Q.ENQUEUE(mc) ;
17 return L' ;
```

*3.4.3 Update Memory Layout.* Algorithm 5 iteratively merges sub-memory layouts by a queue $Q$. The input is a memory layout $L$ and the output is an updated one $L'$. For the memory block $m_q$, it first identifies two sub-memory layouts $L_1$ and $L_2$ (Line 6-7). If their root memory blocks $m_1$ and $m_2$ are intersected (Line 8), we merge them to construct a new memory layout $L_3$ by `MergeBlock` (Line 9). At Lines 10-12, it replaces $L_1$ and $L_2$ with $L_3$. Since the memory layout is updated, we continue to update $L'$ by `UpdateLayout` (Line 13). If sub-memory layouts of $m_q$ cannot be merged, it adds children of $m_q$ into $Q$ (Lines 15-16) and continues to merge the children iteratively. If there is no merging between any two sub memory layouts, the algorithm terminates.

Example 3. Fig. 3 shows the process of merging two memory layouts, which correspond to $L_2$ and $L_3$ in Fig. 2(c). We can infer that $m_1(x_1, 32, 16)$ and $m_2(0, 28, 16)$ are intersected, where $0 \le x_1 \le 12$, $\widetilde{m_1} = \widetilde{m_2} = 16$ and $\overleftrightarrow{m_2} = 28$. Based on Lines 1-3 in Algorithm 4,

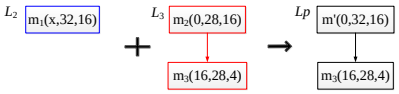



**Figure 3: Example to illustrate merging memory layouts.**

we know $m_1$ and $m_2$ are the alias memory blocks. Hence, they are merged as $m'$=(0, 32, 16), and $m_3$ is added as the inner memory block of $m'$.

*3.4.4 Merging for Pointer Arithmetics.* The pointer arithmetic generally yields a new pointer that still points the same memory allocation [4]. Multiple memory accesses by dereferencing these pointers can lead to many memory layouts, which cannot be merged with Algorithm 2 (because they are not intersected). However, these memory layouts belong to the same memory allocation and should be merged together. To improve the precision, we merge them together as follows:

Given such two memory layouts $L_1 = (m_1, M_1, E_1)$ and $L_2 = (m_2, M_2, E_2)$, they are merged into a new memory layout $L_3 = (m_3, M_3, E_3)$, where $m_3 = (min(m_1^{\llcorner}, m_2^{\llcorner}), max(m_1^{\lrcorner}, m_2^{\lrcorner}), 0)$, $M_3 = M_1 \cup M_2 \cup \{m_3\}$ and $E_3 = E_1 \cup E_2 \cup \{(m_3, m_1), (m_3, m_2)\}$.

Example 4. In Fig. 2(b), Lines $l_1 - l_3$ are executed four times (in the unrolled loop), and we recover four memory layouts, which cannot be merged with Algorithm 2. Due to the pointer arithmetics, these four memory layouts are then merged as $L_1$ in Fig. 2(c).

## 3.5 Discussion of Special Cases

Based on the memory addressing model, our approach can recover memory layouts of variables generally. In this section, we discuss special cases during memory layout recovery.

**Address space layout randomization.** Address space layout randomization (ASLR) is a memory-protection technique by randomizing the locations of modules and certain data [20]. Current ASLR techniques usually randomize the base address of a memory region (e.g., stack frame). In this case, it does not affect our memory layout indexing (c.f. Section 3.4.1) and our approach can still work.

**Memory allocation *alloca*.** The function *alloca* allocates the memory in the stack. In our approach, the stack memory allocated by *alloca* is treated as the heap memory allocation.

**Different memory layouts in one allocation statement.** The layouts of memory allocated in one allocation statement may be different in different contexts. For example, *if(*) size=sizeof(structA); else size=sizeof(structB); ptr=malloc(size);*, the variable *ptr* can represent two data structures (i.e., *structA* and *structB*). For this case, we associate its context with the allocation statement to index the memory layout. Specifically, we adopt the technique in memory indexing [19], which uses control flow structure to index allocation point, as the context of allocation statement.

**Data structure *union*.** A *union* can hold only one of its data members at a time. In multiple accesses of a *union*, it may hold different members. Thus, our approach may merge different members. If different members have different unit sizes, we only recover the maximum memory range and discard the internal memory layout. Hence, the recovered memory layout is still consistent with the semantics of *union*.

# 4 LOCATING VULNERABILITIES

In this section, we introduce how to locate buffer overflow vulnerabilities by leveraging recovered memory layouts.

## 4.1 Locating Vulnerabilities by Leveraging Recovered Memory Layouts

A buffer overflow occurs when dereferencing a pointer that goes out of the bounds of its pointed object. To locate the buffer overflow, we collect a set of passed executions and a failed execution. With the fine-grained memory layouts recovered in passed executions as reference, the vulnerable memory layout in failed execution can be identified by memory layout identification and comparison.

It is worth noting that, the size of a memory block may not be fixed as we introduce the parameters in Algorithm 1 (i.e., $x_i$ at Line 1). To locate the buffer overflow, we adopt a conservative strategy to determine their values: let the size of the passed memory layout be the maximum and the size of the failed memory layout be the minimum. Specifically, in a memory layout $L = (m, M, E)$, the size of each memory block $m_i \in M$ is: $end - (base + \cdots + (x_{i-1} + index_{i-1} * scale_{i-1}) + x_i)$ (c.f. Algorithm 1). For the passed memory layout, we make the size of each $m_i \in M$ maximum, i.e., $(\forall 1 \leq i < n : x_i = 0) \wedge x_n = offset$. For the failed memory layout, we make the size of each $m_i \in M$ minimum, i.e., $x_1 = offset \wedge (\forall 1 < i \leq n : x_i = 0)$.

Algorithm 6 shows how to locate the buffer overflow. It takes a failed memory layout $L_1$ and a passed memory layout $L_2$ as inputs. Its intuition is as follows: for each memory block $m'_1$ of $L_1$, if $m'_1$ is the alias or inner memory block of some memory block $m'_2$ of $L_2$, and $m'_1$ is beyond $m'_2$, then it is a candidate buffer overflow.

At Line 5, it checks whether $\widetilde{m}'_1$ is zero. If so, it cannot determine the relationship between $m'_1$ and $m'_2$. Thus, it only compares $m'_1$ with the root memory block $m_2$ of $L_2$. If $m'_1$ is beyond $m_2$ (Line 6), there is a candidate buffer overflow in $m'_1$. At Line 9-17, it checks $m'_1$ with each memory block $m'_2$ of $L_2$. If $m'_1$ is beyond $m'_2$ (Line 13), the algorithm checks the relationship between $m'_1$ and $m'_2$. If $m'_1$ is the alias or inner memory block of $m'_2$ (Line 14), there is a candidate buffer overflow in $m'_1$.

Notice that, for the heap memory whose size is controlled by inputs, its memory layout is not fixed with different inputs. Although we recover the maximum memory range after merging, it still cannot be used to locate vulnerabilities. For example, in the statement $p = malloc(input)$, if the maximum value of *input* in passed test cases is 5, we recover a memory block whose size is 5 bytes (the maximum). If the value of *input* in the failed test case is 10, we recover a memory block whose size is 10 bytes. Comparing these two memory blocks introduces a false positive. In this case, we dynamically record the memory range of allocated memory, not to index and merge their memory layouts. To locate vulnerabilities, we check whether the used memory is beyond the allocated memory, which is the same as Valgrind Memcheck [3].

## 4.2 False Positive Reduction

Passed executions may only cover partial program behaviors. Thus, the passed memory layout may also be under-approximated. Locating vulnerabilities by comparing failed memory layout with under-approximated memory layout may introduce false positives. For this problem, we reduce false positives based on two accompanying phenomena: data corruption or abnormal memory address [21], which increase the confidence of our results.

**Algorithm 6:** `CompareLayout`

**input** : memory layouts $L_1$=($m_1$,$M_1$,$E_1$) and $L_2$=($m_2$,$M_2$,$E_2$)
**output**: determining the buffer overflow

```
1  Let Q1 be an empty queue ;
2  Q1.ENQUEUE(m1) ;
3  while Q1 is not empty do
4      m'1 ⟵ Q1.DEQUEUE() ;
5      if m̃'1 == 0 then
6          if m'1↳ < m2↳ < m'1↲ or m'1↳ < m2↲ < m'1↲ then
7              return find buffer overflow
8      else
9          Let Q2 be an empty queue ;
10         Q2.ENQUEUE(m2) ;
11         while Q2 is not empty do
12             m'2 ⟵ Q2.DEQUEUE() ;
13             if m'1↳ < m'2↳ < m'1↲ or m'1↳ < m'2↲ < m'1↲ then
14                 if m'1 is alias or inner block of m'2 then
15                     return find buffer overflow
16             foreach mc ∈ Child(m'2) do
17                 Q2.ENQUEUE(mc) ;
18     foreach mc ∈ Child(m'1) do
19         Q1.ENQUEUE(mc) ;
20 return not find buffer overflow
```

*4.2.1 Data Dependence Mismatch.* Buffer overflow typically incurs data corruption (overflowed by another data) [21]. Data corruption can lead to data dependence mismatch, describing a data dependence that is not supposed to exist in the code. For example, in Fig. 2(a), assume that the value of variable $b$ is 4. The program executes four times at Line 12. As a result, the value of variable $ptr \rightarrow stu[a].func$ is corrupted by $ptr \rightarrow stu[a].name$. At Line 14, it uses $ptr \rightarrow stu[a].func$. Thus, there is a data dependence between Lines 12 and 14, which does not exist in the code. Hence, a data dependence mismatch occurs. To increase the confidence, we report a buffer overflow vulnerability only if it conducts a data dependence mismatch as well.

To obtain data dependence relations that do exist in the code, we use dynamic analysis as in work [21]. We execute a set of passed test cases, and compute data dependence relations. Similarly, we also compute the data dependence relations in the failed execution. If a data dependence relation only occurs in failed execution but not in any passed execution, there is a data dependence mismatch.

*4.2.2 Abnormal Memory Address.* When the buffer is overflowed too much, it may reach an memory address that cannot be accessed (e.g., unallocated memory). This situation is considered as an abnormal memory address access. Usually, an abnormal memory access directly leads to a program crash. Thus, if a buffer overflow leads to an abnormal memory address access, it is a true buffer overflow.

# 5 EVALUATION

We have implemented a prototype tool for our approach, and evaluated its effectiveness. All the experiments are performed on the 32-bit Linux system with 3.5 GHz Intel Xeon E5 CPU and 8 GB RAM. Since our memory addressing model is general, our approach can be easily extended to 64-bit system.

## 5.1 Experiment Setup

We selected 25 binary programs from the benchmarks of DARPA's CGC [22], which is a competition to automatically detect vulnerabilities. Instead of contrived simple situations, they approximate real vulnerabilities with enough complexities and diversities, ideal for evaluating our approach [21]. However, not all programs are selected because: 1) they run under DARPA DECREE, while our tool runs on the Linux system. Although the team *TrailofBits* has migrated them into Linux system, not all of them are reproducible [23]; 2) we only consider the buffer overflow, the programs with other types (e.g., null pointer dereference and use after free) are out of our consideration. In addition, we selected four binary programs *objdump*, *readelf*, *ld* and *c++filt* from *binutils* (about 690k LoC), which are widely used in fuzzing system [10, 11], and one binary program *tiff2bw* from *libtiff* (about 100k LoC). In these five programs, we generated 453 program crashes, which constitute real-world benchmarks. The 25 CGC programs show the diversities of vulnerabilities, and the 5 real-world programs show the scalability of our approach.

## 5.2 Experiment Design

In the experiments, we use the fuzzing system [7–11] to generate the passed test cases[2] and use the dynamic binary analysis framework Pin [24] to collect the dynamic execution information. In general, AFL generates a large number of passed test cases with different code coverage. For efficiency, we select the passed test cases by adopting *additional coverage strategy* [25]. It selects the next passed test case, which covers more codes that are covered by the failed test case but not covered by already-selected passed test cases, until vulnerabilities are located. Due to the lack of ground truth in our experiments, we manually validate the results. In total, we manually check 478 program crashes in 30 programs and their recovered memory layouts.

## 5.3 Experimental Results

We evaluated the effectiveness of our approach in three aspects: 1) recovering memory layouts, 2) locating vulnerabilities, and 3) triaging program crashes.

*5.3.1 Recovering Memory Layout.* Table 1 shows the experimental results on recovering memory layouts. Column *Name* lists the program names. In column *Passed Inputs*, the heading *#Total* lists the total number of generated passed test cases, and *#Select* shows the number of selected passed test cases. The details of generation/selection of passed test cases can refer to Section 5.2. Column *Trace Length* lists the number of instructions in the execution, where *#Passed* and *#Failed* represent the average numbers of instructions in selected passed executions and failed execution, respectively.

[2]There could be a situation where a passed execution accesses a memory location beyond the boundary, but does not lead to any crash. This would not produce false positives, not misleading analyst to wrong investigation. See [17] for more details.

Since a program may contain more than one vulnerable memory layout, in column *Vulnerable Memory Layout*, the heading *No.* lists each of them. For example, there are three vulnerable memory layouts in the program *stack_vm*.

The heading *Status* shows the status of recovered passed memory layout, which is used for comparison to identify the vulnerable memory layout. It indicates whether the passed memory layout represents the hierarchical structure of variables, where *Under* means that we under-approximately recover the memory layout, and *Complete* means that we recover its complete memory layouts. Notice that we got a large number of passed memory layouts recovered in passed executions. We manually check and report only 36 of them (column *Status*) because: 1) the number of recovered memory layouts is too large to manually check all; 2) these 36 memory layouts are compared to identify vulnerable memory layouts and we need to check its status (under-approximated or complete).

**Summary.** Among the reported 36 passed memory layouts, 15 memory layouts are completely recovered (i.e., the recovered static hierarchical data structures for variables are the same with their static hierarchical data structures in source code) and 21 are under-approximately recovered (e.g., some internal data structures of variables are not recovered). Memory layouts are under-approximately recovered because some elements of array or members of data structure are not accessed in dynamic execution information. That is, our approach achieves 100 % success rate to recover memory layouts that are covered in dynamic execution information. Despite the under-approximate memory layout recovery, they are still useful to locate buffer overflow vulnerabilities, which is shown in the following experiments.

*5.3.2 Locating Vulnerabilities.* Table 1 also shows the results of locating buffer overflow vulnerabilities. There are multiple types of buffer overflow vulnerabilities in the programs, as shown in column *Type*. The symbols *Stack*, *Heap*, *Global* represent stack, heap, and global buffer overflow, respectively. In addition, we consider another special type of buffer overflow: overflow within a data structure, as indicated by *Internal*. In binary code, Valgrind Memcheck [3] cannot locate buffer overflows within stack, global memory regions, and data structures [1]. Valgrind's extension SGCheck tries to locate stack buffer overflows, however, it still needs debug information. Even in source code, AddressSanitizer [1] cannot detect the buffer overflow within the data structures as well.

Column *Buffer Overflow* shows whether the vulnerable memory layouts are the root causes of crash. Since one vulnerable memory layout may be overflowed at different instructions, the heading *#Ins* represents the number of instructions producing buffer overflows in vulnerable memory layouts. The heading *Root* shows whether it is a real buffer overflow. Our approach may report false positives. To reduce them, we adopt two strategies: data dependence mismatch (denoted by M) and abnormal memory address (denoted by A) in the column *Plus Accompanying*.

After our investigation, we found that the false positives are generated in two cases: (1) Some instructions do lead to the buffer overflow, but do not lead to the crash. For example, in the program *Sample_Shipgame*, there are 2 instructions leading to a buffer overflow. However, one is assigned with '\0', and it is not the root cause of crash. This can be eliminated by our strategies, and thus

**Table 1: Experimental results on programs from DARPA's Cyber Grand Challenge**

| Name | Type | Passed Inputs | | Trace Length | | Vulnerable Memory Layout | | Buffer Overflow | | Plus Accompanying | | | Time(s) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #Total | #Select | #Passed | #Failed | No. | Status | #Ins | Root | Mismatch/Abnormal | #Ins | Root | Select | Locate |
| Sample_Shipgame | Stack | 620 | 1 | 20,843 | 32,502 | 1 | Under | 2 | ✓ | M | 1 | ✓ | 349 | 50 |
| ValveChecks | Stack | 10 | 2 | 5,242 | 104,726 | 1 | Under | 1 | × | – | 0 | – | 6 | 51 |
| | | | | | | 2 | Complete | 1 | ✓ | M+A | 1 | ✓ | | |
| Bloomy_Sunday | Stack | 192 | 1 | 786,145 | 642,941 | 1 | Complete | 2 | ✓ | M+A | 1 | ✓ | 119 | 147 |
| The_Longest_Road | Stack | 298 | 1 | 343,992 | 344,668 | 1 | Under | 3 | ✓ | M+A | 1 | ✓ | 168 | 110 |
| Thermal_Controller_v2 | Stack | 83 | 1 | 64,752 | 66,291 | 1 | Under | 5 | ✓ | M | 3 | ✓ | 52 | 104 |
| XStore | Stack | 290 | 1 | 995,805 | 997,222 | 1 | Under | 1 | × | – | 0 | – | 163 | 387 |
| | | | | | | 2 | Under | 3 | ✓ | M+A | 1 | ✓ | | |
| Casino_Games | Stack | 603 | 2 | 192,211 | 182,209 | 1 | Under | 1 | ✓ | M+A | 1 | ✓ | 390 | 143 |
| Palindrom | Stack | 61 | 1 | 106,087 | 5,262 | 1 | Complete | 6 | ✓ | M | 4 | ✓ | 31 | 48 |
| CableGrind | Stack | 818 | 1 | 17,240 | 18,336 | 1 | Under | 1 | ✓ | M+A | 1 | ✓ | 432 | 46 |
| stack_vm | Heap | 168 | 1 | 177,047 | 1,104,414 | 1 | Under | 1 | × | – | 0 | – | 95 | 378 |
| | | | | | | 2 | Under | 1 | ✓ | A | 1 | ✓ | | |
| | | | | | | 3 | Complete | 1 | ✓ | M | 1 | ✓ | | |
| Street_map_service | Heap | 630 | 1 | 1,121,017 | 714,785 | 1 | Complete | 3 | ✓ | M | 1 | ✓ | 468 | 240 |
| humaninterface | Heap | 533 | 1 | 509,169 | 509,618 | 1 | Complete | 1 | ✓ | M | 1 | ✓ | 279 | 244 |
| AIS-Lite | Heap | 368 | 1 | 41,069 | 33,904 | 1 | Under | 3 | × | – | 0 | – | 203 | 47 |
| | | | | | | 2 | Complete | 2 | ✓ | M | 1 | ✓ | | |
| matrices_for_sale | Heap | 34 | 1 | 170,680 | 6,564 | 1 | Complete | 1 | ✓ | A | 1 | ✓ | 18 | 54 |
| cotton_swab_arithmetic | Heap | 1065 | 1 | 2,192 | 2,052 | 1 | Complete | 1 | ✓ | A | 1 | ✓ | 544 | 37 |
| LMS | Heap | 147 | 1 | 26,710 | 22,425 | 1 | Complete | 2 | ✓ | M | 1 | ✓ | 81 | 58 |
| BudgIT | Heap | 222 | 1 | 169,483 | 9,502 | 1 | Under | 2 | × | – | 0 | – | 123 | 72 |
| | | | | | | 2 | Complete | 2 | ✓ | A | 1 | ✓ | | |
| PKK_Steganography | Heap | 184 | 1 | 182,333 | 114,543 | 1 | Complete | 1 | ✓ | A | 1 | ✓ | 102 | 178 |
| ASCII_Content_Serve | Heap | 277 | 1 | 1,377,145 | 489,714 | 1 | Complete | 1 | ✓ | A | 1 | ✓ | 174 | 551 |
| electronictrading | Internal | 90 | 2 | 595,381 | 10,800 | 1 | Under | 2 | ✓ | M | 1 | ✓ | 49 | 186 |
| SCUBA_Dive_Logging | Internal | 699 | 2 | 706,641 | 83,424 | 1 | Under | 3 | × | – | 0 | – | 477 | 303 |
| | | | | | | 2 | Under | 2 | × | – | 0 | – | | |
| | | | | | | 3 | Under | 1 | ✓ | M | 1 | ✓ | | |
| CGC_Planet_Mark_ Language_Parser | Internal | 583 | 6 | 999,709 | 131,256 | 1 | Under | 3 | × | – | 0 | – | 612 | 1,432 |
| | | | | | | 2 | Under | 5 | × | – | 0 | – | | |
| | | | | | | 3 | Under | 3 | × | – | 0 | – | | |
| | | | | | | 4 | Under | 6 | × | – | 0 | – | | |
| Square_Rabbit | Global | 593 | 1 | 2,917,848 | 837,835 | 1 | Complete | 1 | ✓ | A | 1 | ✓ | 484 | 605 |
| TAINTEDLOVE | Global | 52 | 1 | 201,339 | 100,849 | 1 | Under | 1 | ✓ | A | 1 | ✓ | 29 | 73 |
| stream_vm | Global | 136 | 1 | 101,184 | 100,145 | 1 | Complete | 1 | ✓ | A | 1 | ✓ | 67 | 76 |
| Total | 4 | 8,756 | 34 | 11,831,264 | 6,665,987 | 36 | 15+21 | 76 | 25+11 | 25+11 | 30 | 25+0 | 5,515 | 5,620 |
| Avg. | – | 350 | 1.4 | 473,251 | 266,639 | – | – | 3.04 | – | – | 1.2 | – | 221 | 225 |

the number of instructions is reduced from 2 (*#Ins* in *Buffer Overflow*) to 1 (*#Ins* in *Plus Accompanying*). (2) Some instructions indeed do not produce the buffer overflow. For example, in the program *CGC_Planet_Mark_Language_Parser*, all the false positives are reduced (*#Ins* in *Plus Accompanying*).

Since our approach requires passed and failed executions, its effectiveness depends on the test cases. For example, in program *CGC_Planet_Markup_Language_Parser*, our approach fails to locate buffer overflow vulnerabilities. This is because the program contains many special checks in the markup language parser, and AFL does not generate passed test cases to cover the data structure overflowed (i.e., struct *City*). Since our approach does not recover the memory layout of struct *City* in passed executions, we fails to locate its buffer overflow in failed execution.

Column *Time* shows the time overhead. The heading *Select* shows the time for selecting passed test cases, while *Locate* for locating buffer overflow vulnerabilities (including recovering memory layouts, computing data dependencies, and locating vulnerabilities).

**Summary.** Our experimental programs include 4 types of vulnerabilities (Column *Type*): 9 stack buffer overflows, 10 heap buffer overflows, 3 internal data structure overflows, 3 global memory buffer overflows, which show the diversity of vulnerabilities. For each vulnerable memory layout, we locate 3.04 instructions on average. In the beginning, we reports 25 true positives and 11 false positives (*Root* in *Buffer Overflow*). After applying the proposed elimination strategies, 25 errors are confirmed, and all of 11 false positives are eliminated (*Root* in *Plus Accompanying*). The average time for selecting passed test cases and locating vulnerabilities are 221 and 225 seconds. The results show that our approach is effective to locate buffer overflow vulnerabilities.

**Table 2: Triage Program Crash on five real-world programs**

| Name | Test Cases | | Trace Length | | AFL | Our Approach |
|---|---|---|---|---|---|---|
| | #passed | #failed | #Passed | #Failed | | |
| objdump-2.26 | 1,475 | 73 | 166,802 | 235,657 | 59 | 4 (61, 8, 3, 1) |
| readelf-2.28 | 1,780 | 119 | 162,662 | 68,088 | 69 | 3 (102, 15, 2) |
| ld-2.24 | 1,274 | 117 | 1,504,274 | 1,334,977 | 90 | 6 (45, 28, 12, 12, 3, 1) |
| c++filt-2.26 | 1,861 | 23 | 17,708 | 4567 | 18 | 3 (21, 1, 1) |
| tiff2bw-3.9.7 | 1,846 | 121 | 1,131,200 | 1,102,423 | 84 | 3 (111, 6, 4) |

*5.3.3 Triage Program Crashes.* As described in Section 1, triaging program crashes is very important in the program analysis and debugging. In the experiments, we use the fuzzing system to generate a number of program crashes, many of which are caused by the same vulnerability but AFL is not able to distinguish them. Our

approach can help to triage program crashes based on the root causes of vulnerabilities.

Since multiple instructions may lead to the same vulnerable memory layout (e.g., *#Ins* in *Buffer Overflow*), we decided to use a more coarse-grained granularity, i.e., we use functions where blamed instructions reside as blamed functions. Thus, in our experiment, we identify the root causes based on vulnerable memory layouts and blamed functions [6].

Table 2 shows the results of five real-world programs with total 453 program crashes. Column *Name* shows the program name. Column *Test Cases* lists the number of test cases, where *#Passed* and *#Failed* represent the number of passed and failed test cases, respectively. Column *Trace Length* lists the average number of instructions in the execution. The last two columns show the results of triaging program crashes.

**Summary.** Column *AFL* lists the number of unique program crashes triaged by AFL. Column *Our Approach* shows triaging results of our approach, where the first number is the number of groups, and the numbers in the brackets represent the number of crashes in each group. For example, $4(61, 8, 3, 1)$ represents 73 crashes are classified into 4 groups, which include 61, 8, 3 and 1 crashes, respectively. We manually check the crashes for all groups and confirm that our triaging results are correct. Compared to AFL, our approach helps to reduce the number of unique program crashes significantly. Thus, it can save manual resources for analyzing them.

There are 16 crashes that are not grouped in program *ld-2.24*, as our approach fails to locate their vulnerabilities. We look closely into this program, and these 16 crashes are due to the null pointer dereference, which is out of the scope of this work.

# 6 RELATED WORK

## 6.1 Reverse Engineering

Reverse engineering of data structures is an active area in binary code analysis [39–43]. Thomas and Gogul [12] also proposed an approach to recover memory ranges of variables in binary code. Its main difference from ours is that they recover memory information using static analysis (value-set analysis), while we use dynamic execution information. Our results are under-approximated and theirs are over-approximated. Our under-approximated memory layout introduces less false negatives and false positives in locating buffer overflow vulnerabilities. Built on the work [12], Brumley *et al.* developed TIE [13], which recovers the memory information of data structures and their types. TIE has the same limitations as work [12] when used for locating buffer overflow vulnerabilities.

Lin *et al.* proposed Rewards [14], the reverse engineering of data structures using dynamic analysis. It infers the types of data structures based on arguments of well-known functions (e.g., a system call). However, our approach recovers memory layouts based on the memory addressing model. Hence, Rewards only recovers a small portion of data structures [15], and our approach is more general for data structures.

Slowinska *et al.* developed Howard [15, 16], which is the closest work to ours. However, there are still two main differences. The first is that Howard may miss internal layouts of data structures in some cases even they are accessed in the execution. Howard [15] only records a base pointer for each memory access. It may miss the internal layouts in the case where memory address is computed based on multiple base addresses. Take the memory access at Line 12 in Fig 2 for example, Howard considers its base pointer is *ptr*. However, its memory address is computed based on *ptr*, *stu* and *name*. Hence, Howard misses the internal layouts. The second difference is that Howard detects arrays which are accessed in loops while our approach can recover arrays generally based on the memory addressing model. Thus, our approach is more general to recover the fine-grained layouts of data structures.

## 6.2 Locating Vulnerabilities

**Source code analysis.** There has been lots of work aiming at locating vulnerabilities in source code (e.g.,[53–56]). AddressSanitizer [1] is a widely used tool in practice. It instruments a program, inserts undefined memory (i.e., redzones) between the objects and detects an access to the undefined memory. DieHard [53] and its successor DieHarder [54] populate newly allocated memory and freed memory with magic values. They also add redzones around the allocated memory region to detect the spatial errors. The tools SoftBound [2], CETS [4] and LowFat [55, 56] keep track of per-pointer capability and checks capability when accessing an object.

Although these techniques can also locate buffer overflows, they are applicable to different scenarios (i.e., source code *vs.* binary code). Since the source code is not always available, our approach is applicable to more scenarios. In addition, some source code based techniques do not recognize the internal structures of data structures, such as AddressSanitizer. Hence, they cannot locate internal overflow within a data structure. Since our approach recovers fine-grained memory layouts of variables, this is not an issue anymore.

**Binary code analysis.** Locating vulnerabilities in binary code has also been widely studied [57–62]. Valgrind Memchecks [3] uses the valid value bit and address bit in shadow memory to capture reading undefined memory and out-of-bounds access. Besides, Valgrind's extension SGCheck also wants to locate stack buffer overflows, however, it still needs the help of debug information. Dr. Memory [59] is similar to Valgrind Memchecks in many ways. It is further equipped with a multi-threaded binary translation system. Purify [60] shadows every byte of memory with a two-bit value that encodes one of three states: unaddressable, writable, and readable.

These techniques locate heap buffer overflow without false positives, which is achieved by our approach as well. Due to the lack of program semantics in binary code, it is very difficult to identify the boundaries of variables in stack and global memory regions. Thus, none of these techniques can locate buffer overflow within the stack and global memory regions [1]. To the best of our knowledge, our approach is the first work to achieve this goal in binary code.

# 7 CONCLUSION

In this work, we propose an approach to locate buffer overflows in binary code. We first recovers the memory layouts based on memory addressing model together with dynamic execution information. Then, based on the recovered memory layouts we locate buffer overflow vulnerabilities.

# REFERENCES


[1] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[2] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[3] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[4] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[5] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 24–35.

[6] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "Retracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 820–831.

[7] AFL, "American fuzzy lop. http://lcamtuf.coredump.cx/afl/."

[8] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 254–265.

[9] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 475–485.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[11] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[12] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[13] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[14] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 2010, p. 5.

[15] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *Proceedings of the Network and Distributed System Security Symposium , year=2011, organization=Citeseer*.

[16] ——, "Body armor for binaries: Preventing buffer overflows without recompilation." in *USENIX Annual Technical Conference*, 2012, pp. 125–137.

[17] "Proof," https://sites.google.com/site/memorylayout/proof.

[18] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 103–114, 2008.

[19] W. N. Sumner and X. Zhang, "Memory indexing: canonicalizing addresses across executions," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 217–226.

[20] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[21] Y. Chen, M. Khandaker, and Z. Wang, "Pinpointing vulnerabilities," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 334–345.

[22] DARPA, "Cyber grand challenge repository. https://github.com/cybergrandchallenge/."

[23] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[25] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *2016 IEEE/ACM 38th International Conference on Software Engineering*. IEEE, 2016, pp. 535–546.

[26] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2016.

[27] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 192–201.

[28] S. A. Khalek and S. Khurshid, "Efficiently running test suites using abstract undo operations," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 110–119.

[29] W. Masri and F. A. Zaraket, "Coverage-based software testing: Beyond basic test requirements," in *Advances in Computers*. Elsevier, 2016, vol. 103, pp. 79–142.

[30] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *2013 35th International Conference on Software Engineering*. IEEE, 2013, pp. 622–631.

[31] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Acm Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 504–515.

[32] J. H. Siddiqui and S. Khurshid, "Staged symbolic execution," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1339–1346.

[33] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *ACM SigPlan Notices*, vol. 48, no. 10. ACM, 2013, pp. 19–32.

[34] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *IEEE/IFIP International Conference on Dependable Systems & Networks*. Citeseer, 2009, pp. 359–368.

[35] B. C. Parrino, J. P. Galeotti, D. Garbervetsky, and M. F. Frias, "Tacoflow: optimizing sat program verification using dataflow analysis," *Software & Systems Modeling*, vol. 14, no. 1, pp. 45–63, 2015.

[36] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.

[37] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[38] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 134–138.

[39] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 1–28.

[40] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures." in *USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 255–266.

[41] E. Dolgova and A. Chernov, "Automatic reconstruction of data types in the decompilation problem," *Programming and Computer Software*, vol. 35, no. 2, pp. 105–119, 2009.

[42] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *Proceedings of the Network and Distributed System Security Symposium*. Citeseer, 2015.

[43] V. Braberman, D. Garbervetsky, S. Hym, and S. Yovine, "Summary-based inference of quantitative bounds of live heap objects," *Science of Computer Programming*, vol. 92, pp. 56–84, 2014.

[44] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[45] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Automated software engineering*, vol. 17, no. 1, p. 5, 2010.

[46] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 515–524.

[47] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 56–66.

[48] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 73–84.

[49] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of Programming Language Design and Implementation*, 2005, pp. 15–26.

[50] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2017, pp. 114–125.

[51] X. Li, M. dAmorim, and A. Orso, "Iterative user-driven fault localization," in *Haifa Verification Conference*. Springer, 2016, pp. 82–98.

[52] A. Perez, R. Abreu, and M. d'Amorim, "Prevalence of single-fault fixes and its impact on fault localization," in *2017 IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2017, pp. 12–22.

[53] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Acm sigplan notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.

[54] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 573–584.

[55] G. J. Duck and R. H. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 132–142.
[56] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *Symposium on Network and Distributed System Security*, 2017.
[57] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries." in *USENIX Security Symposium*, 2016, pp. 583–600.
[58] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications." in *USENIX Security Symposium*, 2014, pp. 829–844.
[59] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 213–223.
[60] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the 1992 USENIX Conference*. Citeseer, 1991.
[61] Intel, "Intel inspector. https://software.intel.com/en-us/intel-inspector-xe/."
[62] Oracle, "Sun memory error discovery tool. https://docs.oracle.com/cd/e18659_01/html/821-1784/gentextid-302.html."