


RESEARCH

Open Access



# Memory access integrity: detecting fine-grained memory access errors in binary code

Wenjie Li<sup>1,3,4,5\*</sup> , Dongpeng Xu<sup>2</sup>, Wei Wu<sup>1,3,4,5</sup>, Xiaorui Gong<sup>1,3,4,5</sup>, Xiaobo Xiang<sup>1,3,4,5</sup>, Yan Wang<sup>1,3,4,5</sup>, Fangming gu<sup>1,3,4,5</sup> and Qianxiang Zeng<sup>1,3,4,5</sup>

## Abstract

As one of the most notorious programming errors, memory access errors still hurt modern software security. Particularly, they are hidden deeply in important software systems written in memory unsafe languages like C/C++. Plenty of work have been proposed to detect bugs leading to memory access errors. However, all existing works lack the ability to handle two challenges. First, they are not able to tackle fine-grained memory access errors, e.g., data overflow inside one data structure. These errors are usually overlooked for a long time since they happen inside one memory block and do not lead to program crash. Second, most existing works rely on source code or debugging information to recover memory boundary information, so they cannot be directly applied to detection of memory access errors in binary code. However, searching memory access errors in binary code is a very common scenario in software vulnerability detection and exploitation.

In order to overcome these challenges, we propose Memory Access Integrity (MAI), a dynamic method to detect fine-grained memory access errors in off-the-shelf binary executables. The core idea is to recover fine-grained accessing policy between memory access behaviors and memory ranges, and then detect memory access errors based on the policy. The key insight in our work is that memory accessing patterns reveal information for recovering the boundary of memory objects and the accessing policy. Based on these recovered information, our method maintains a new memory model to simulate the life cycle of memory objects and report errors when any accessing policy is violated. We evaluate our tool on popular CTF datasets and real world softwares. Compared with the state of the art detection tool, the evaluation result demonstrates that our tool can detect fine-grained memory access errors effectively and efficiently. As the practical impact, our tool has detected three 0-day memory access errors in an audio decoder.

**Keywords:** Binary analysis, Fine-grained, Memory access error, Detection

## Introduction

Memory access errors, e.g., stack/heap overflow, use after free, use before initialization, have been the most dangerous software vulnerabilities. A successful exploit (Chen et al. 2005) of memory access error may lead to arbitrary code execution or leak of sensitive data. These errors usually hide in critical component of software systems written in memory unsafe languages such as C/C++. They are easy to be neglected but

severely threaten modern software security. In 2017, users are still reporting blue screen errors caused by “ATTEMPTED\_EXECUTE\_OF\_NOEXECUTE\_MEMORY” when using Windows operating system (Maklakov 2017).

To tackle memory access errors, researchers have proposed various methods to detect them in software systems. One category of detection methods (Serebryany et al. 2012; Nagarakatte et al. 2009; Oleksenko et al. 2017) check out-of-bounds memory access and dereference of dangling pointers by leveraging source code level information (e.g., type information) and compiler assisted instrumentation. Another category is binary level memory error detector, such as Valgrind’s memcheck plugin (Nethercote and Seward 2007b) and Dr. Memory (Bruening and Zhao

\*Correspondence: [liwenjie@iie.ac.cn](mailto:liwenjie@iie.ac.cn)

<sup>1</sup>School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

<sup>3</sup>Key Laboratory of Network Assessment Technology, CAS, Beijing, China  
Full list of author information is available at the end of the article

2011). They recover coarse-grained memory boundary (i.e., the size of memory chunk returned by `malloc`), and enforces a set of security policies to detect various memory access errors (e.g., stack overflow, heap overflow, use after free, use before initialization).

Unfortunately, existing detection methods suffer from several limitations. First, these methods only check coarse-grained memory access error, but they are not able to detect memory access errors inside one memory chunk. Particularly, much complex software includes their own memory management module, which usually claims a large memory chunk from operating system and then organizes their own data structure inside the memory chunk. Existing coarse-grained methods can only detect memory access errors across the outermost memory chunk boundary. They cannot handle memory access errors happened inside data structures within one chunk. The fine-grained memory access information is critical for locating where the memory access error happens and how to fix it. Second, source code or debugging information is missing or not available in many scenarios in practice, e.g., when detecting vulnerabilities in third-party software or checking legacy software. Existing methods utilizing source code information do not directly work on binary code. Moreover, existing methods using compiler assisted instrumentation (Lattner and Adve 2004) introduce memory layout differences, which results in false negatives or the error location is not accurately reported.

To overcome the challenges above, we propose a novel method called “Memory Access Integrity” (MAI) for detection of fine-grained memory access errors in binary code without any source code or debugging information. Our method tracks memory access patterns during runtime. Based on the memory accessing information, it infers and maintains accessing policy between pointers and memory objects. A warning is reported when a memory access behavior conflicts with the rule.

The key insight is that a boundary of memory objects and accessing policy can be inferred from instructions by checking memory access patterns in binary code during runtime. To be more specific, our method recognizes the “base+offset” memory access pattern, which provides strong evidence of the boundary and accessing policy. Our method maintains a new memory model, Memory Range Record, to describe the boundary and relation of memory objects during the whole program execution. It reports a memory access error when an instruction tries to access a memory address that is out of the memory boundary controlled by that based address.

Compared with existing methods, our approach facilitates detection of memory access errors from the following aspects. First, the memory access policy is collected via memory access patterns, which naturally reflect the data structures inside memory. Therefore, our method

has the ability to handle fine-grained data structures. Second, the inference and checking of memory access policy are purely on assembly instructions, so our method can directly analyze binary code, requiring no help from source code or compilers. To the best of our knowledge, our method is the first one that can detect fine-grained memory access errors in binary code.

To demonstrate the effectiveness of our method, we apply MAI to a set of CTF challenges containing different categories of memory access errors and MAI successfully detects all memory errors. We also apply MAI to real world programs and compare the detection result with Valgrind, the state of the art. Our result shows that MAI effectively and efficiently detects all fine-grained and coarse-grained memory access errors. Particularly, MAI’s practical impact is demonstrated by finding three 0-day memory access errors in an audio decoder.

In summary, this paper makes the following contributions.

- We propose Memory Access Integrity (MAI), a novel method for checking fine-grained memory access errors in binary code. Our method infers memory access policy based on memory access patterns, and then check memory access errors during program execution.
- We have implemented MAI as a prototype tool and include it in a cross-platform binary analysis framework for detecting and exploiting memory corruption vulnerability.
- We evaluate MAI on various scenarios including CTF challenges and real world software. The result demonstrates MAI can effectively detect and diagnose memory access errors. The practical impact of MAI is demonstrated by the detection of three 0-day vulnerabilities in an audio decoder.

The rest of the paper is organized as follows. Section [Background](#) describes the background and challenges. Section [Overview](#) presents the overview of MAI. Section [Memory range record-Error detection](#) describes the design of our method in detail. “[Implementation](#)” section presents the implementation and “[Evaluation](#)” section describes the evaluation result. We summarize and discuss related work in Section [Related work](#), and finally conclude this work in Section [Conclusion](#).

## Background

### Memory access error

A memory access error occurs when a program tries to access an illegal memory location. Common memory access errors include: write across boundary, read uninitialized memory, use after free, double free. These errors are widely hidden in important software systems written in memory unsafe programming languages such as C/C++.

Figure 1 shows an example of write across boundary, a common memory access error. When the function `fun` copies `buf` to array `a[10]`, no boundary checking is performed, so the integer variable `flag` could be overflowed.

### Detection of memory access error

Memory access errors are not only bugs that may lead to a program crash, but also severe vulnerabilities that could be exploited by attackers. Many attacking methods like ROP (Checkoway et al. 2010; Roemer et al. 2012; Buchanan et al. 2008) rely on memory access errors, such as buffer overflow or dangling pointers, to trigger the first step. Therefore, researchers have been working on various detection methods to check memory access errors in software.

One group of detection methods aim to help software developers find and correct memory access errors in software development, so these methods require support from compilers and other tool chains. AddressSanitizer (ASan) (Serebryany et al. 2012) is an open source compiler extension developed by Google. It is based on redzone instrumentation in compilers. Redzone is a technique that adds various types of special memory segment between memory areas. When out-of-bounds access happens, the memory operation will first access the redzone memory areas, which will trigger a warning. In the hardware field, Intel MPX (Memory Protection Extensions) (Oleksenko et al. 2017) is a set of extensions to the x86 instruction set architecture. Intel MPX brings increased security to software by checking pointer references. It checks if pointer references cause a buffer overflow at runtime. MPX can detect the intra-object-overflow vulnerability, but it also needs source code. WIT (Akritidis et al. 2008) uses points-to analysis at compile time to compute the control flow graph and the set of objects that can be written by each instruction in the program. Then it generates code instrumented to prevent instructions from modifying objects that are not in the set computed by the static analysis.

```
1 struct A {
2     char a[10];
3     int flag;
4 };
5 void fun ()
6 {
7     struct A x;
8     ...
9     strcpy(x.a, buf); // overflow
10    ...
11 }
```

**Fig. 1** An example of memory access error

Softbound (Nagarakatte et al. 2009) is a compile-time transformation for enforcing spatial safety of C. It records base and bound information for every pointer as disjoint metadata based on the static analysis of the source code. However, all these methods rely on source code information. They are not suitable for analyzing binary code.

Another group aims to detect memory access errors in a binary environment. These methods get necessary information from binary and do not need source code assistance. Memcheck plugin of Valgrind (Nethercote and Seward 2007b) aims to recover memory bounds at runtime and enforce a set of security policies to detect various memory corruption bugs. Memcheck monitors heap memory allocation/deallocation to infer the bounds between heap chunks. Valgrind also leverages *redzone* technology to do memory error detection. However, Valgrind only inserts redzones between coarse-grained memory chunks, so it is not able to detect fine-grained memory access errors inside one memory chunk.

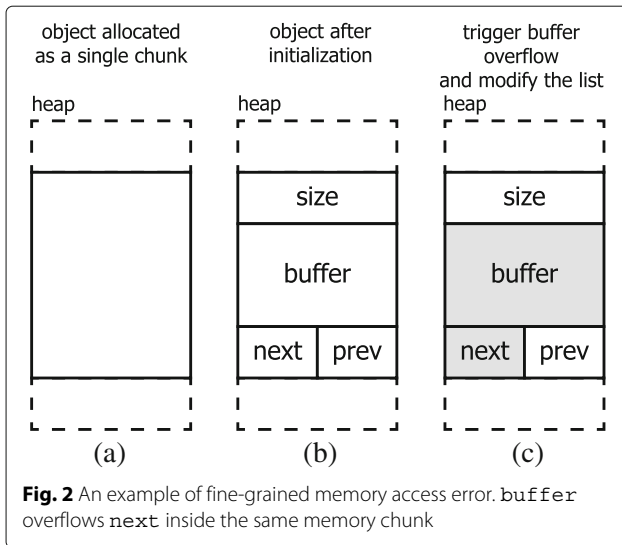
Dr. Memory (Bruening and Zhao 2011) is a method introducing a feature called “nudge” to allow users to request a leak scan at any point. It mainly utilizes shadow memory to maintain the status of each memory bytes and cannot detect fine-grained memory access errors.

### Fine-grained memory access error

A fine-grained memory access error happens when a member variable inside a data structure is overflowed. An attacker can use this overflow vulnerability to control another member variable to exploit the program. Usually, this type of overflow does not exceed the memory chunk boundary and does not lead to a program crash.

Figure 2 illustrates a fine-grained memory access error in a link list node. First, a large memory chunk is allocated for storing one link list node. Next, member variables are initialized as shown in Fig. 2b. The object has a buffer which is prone to be overflowed. When an overflow happens, the out-of-bounds-write will write in other members such as `next`. If the overflowed member variable can be exploited to control program execution, the vulnerability is very dangerous.

Because fine-grained memory access errors happen inside data structures, detection of this type of vulnerability requires more information about the data structures inside memory chunks. Unfortunately, because this information is implicit and challenging to be recovered from binary code, many existing works including Valgrind only do coarse-grained analysis (Castro et al. 2006; Jim et al. 2002; Nagarakatte et al. 2009; Dhurjati et al. 2006). Other existing methods (Austin et al. 1994; Dhurjati and Adve 2006; Condit et al. 2007; Lam and Chiueh 2005; Nacula et al. 2005; Patil and Fischer 1997; Xu et al. 2004; Yong and Horwitz 2003) rely on type information from source



code to do fine-grained detection. These methods are still not feasible for detection of fine-grained memory access errors in binary code.

### Challenges

For detection of fine-grained memory access errors as shown in Fig. 2c in binary code, necessary information regarding boundaries of memory ranges have to be collected. The most important challenges are summarized as follows.

**Missing boundary.** To detect out-of-bounds memory write, the boundary information for memory access operations is necessary. Although it is possible to build up some coarse-grained boundary for memory chunks by tracking heap allocation and deallocation, the accurate boundaries that separate sub-fields inside a data object is

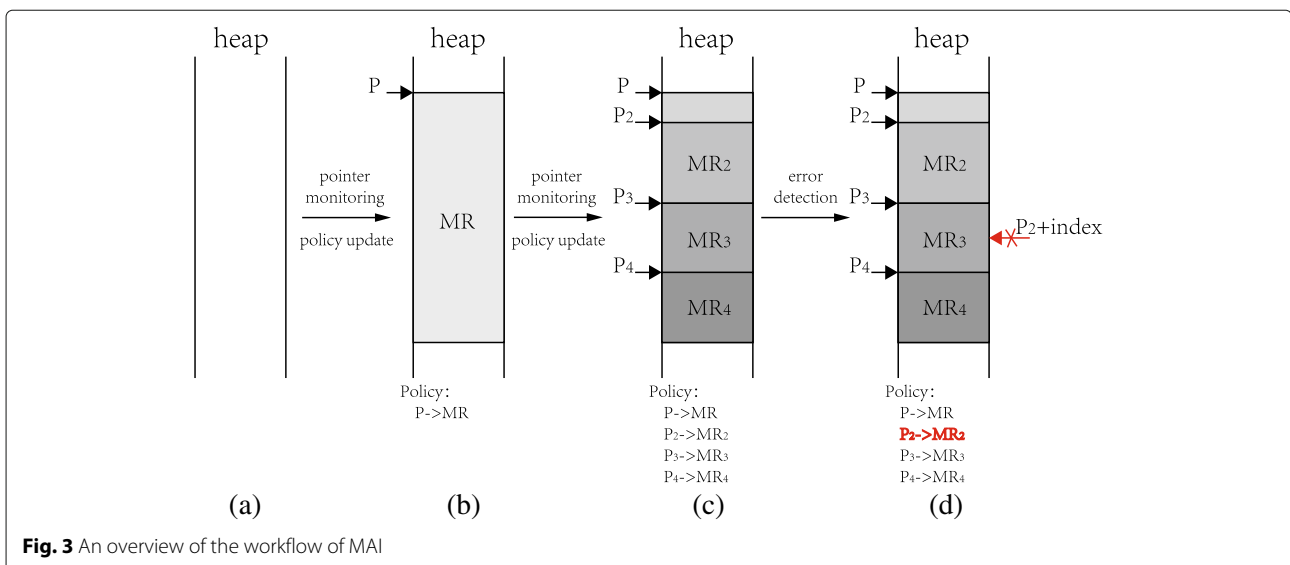
still missing. For example, the boundary information of buffer and list node in Fig. 2c is lost in the binary code.

**Lack of pointer information.** Suppose the necessary boundary information is available, it still remains challenging to judge whether a pointer based memory access is valid due to the lack of pointer information, e.g., which pointer is valid for accessing a memory range. In other words, we need an accessing policy to check whether a pointer is legally accessing memory blocks.

### Overview

In order to overcome the limitations, we propose Memory Access Integrity (MAI), a novel dynamic method for detection of fine-grained memory access errors in off-the-shelf binaries. Our method monitors memory access behaviors during runtime and reports warnings when illegal memory access happens. This is done by building and maintaining a fine-grained memory model to describe the relation between pointers (memory access location) and the range of memory objects. Figure 3 shows a typical step-by-step workflow of our method.

As shown in Fig. 3a, no memory chunk is allocated on the heap at the beginning of program execution. Next, when the program requests a memory chunk and the address is stored in pointer  $P$ , we create a policy showing  $P$  is feasible to access the range of the memory range  $MR$  as shown in Fig. 3b. When the program uses  $P$  to access a member variable inside this memory chunk, we record the new memory access address and the related memory range. As shown in Fig. 3c,  $P_2, P_3, P_4$  point to different memory sub-regions inside this memory chunk and different gray blocks represent the memory range they can access. The policy is also updated accordingly. In Fig. 3d, when the pointer  $P_2$  is used for out-of-bound accessing the memory block, it violates the policy so our system reports a memory access error.



## Memory access integrity

### Design principle

As the key method in our work, we introduce the concept of “Memory Access Integrity” (MAI) to describe and check whether a memory access is legal or not. The key idea is inferring the boundary of memory objects from the relation between pointers and memory chunks on the fly. With this information, we recover the memory access policy and use it to check memory access behaviors.

Memory Access Integrity is guaranteed by policy checking. More specifically, policy in this paper refers to a set of rules. Each rule is a pair defined as follows. It restricts a memory range to be legally accessed by any pointers inside a pointer set.

**Definition 1** A rule is a pair  $(PS, MRR)$ , where  $PS$  is a set of pointers and  $MRR$ , Memory Range Record, is a memory model for describing memory range and state.

The rules in a policy are inferred from program execution. We achieve this by monitoring pointers dynamically, maintaining a new memory model to describe the boundary and life cycle of memory objects, and inferring the relation between pointers and the memory model. Before digging into the details of how we complete these tasks in our method, we provide an intuitive example to walk through these steps. The example presents the following two key insights into our method.

- 1 The first write operation associated with a pointer can help us know the “point-to” relation between a pointer and a memory chunk.
- 2 The “base+offset” addressing scheme can be leveraged to infer the ownership between memory chunk and pointers.

### An example for walk-through MAI

We use a concrete example to quickly walk through MAI. Figure 4 presents the source and assembly code related to the motivating example in Fig. 2.

The example shows operations on a linked list. We will elaborate on how we build the relation between pointers and memory chunks and the relation between memory chunks. At line 7 of the left hand side of Fig. 4, the program allocates a node object and we obtain its address and size. From the instruction at Line 9, the node pointer is used as the base address to calculate the address of `node->next` and write 8 bytes in this space. So we know that an 8-byte sub-region begins at the offset 0x88 is used in this memory chunk. Similarly, at line 11, the node pointer is used as the base address to get its sub-region, `buffer`. Figure 5 shows the relation between pointers and memory chunks and the relation between memory chunks after the program execution.

## Memory range record

As the core part to support memory access integrity, we design a new memory model called “Memory Range Record” (MRR) to record the fine-grained boundary and life cycle of memory objects. In this section, we first present the concept of MRR and then elaborate various types of actions on MRR, including generation, range setting, and deletion. Specifically, we introduce how we gradually divide a memory chunk into sub-regions in a fine-grained way based on the memory access pattern of executed instructions.

### MRR and MRR tree

In this paper, a Memory Range Record (MRR) is a tuple defined as follows. It describes a memory range and the state of the range. For every MRR, we assign a unique ID number.

**Definition 2** A MRR is a tuple  $(id, start, end, state)$ , where  $id$  is a unique number,  $start$  is the starting address of the range,  $end$  is the ending address of the range, and  $state$  is one of *uninit*, *used*, or *free*.

MRR is constructed during program execution. In general, our method creates a new MRR by capturing two types of memory access patterns as follows. These patterns reveal information such as the location of memory objects and pointers to access them. More details of MRR generation is elaborated in Section [MRR generation](#).

- 1 A new memory chunk is allocated, e.g., `malloc` is called.
- 2 A memory object is accessed by `base+offset` pattern.

In real world software, memory access patterns are complex. A very common situation is data structures, which usually have various levels of nested memory ranges. In order to tackle this problem, we organize MRRs as a tree structure. The child MRR is a sub-range of its parent MRR. Our system generates child MRRs by capturing the `base+offset` pattern. When the `base` pointer is the starting address of the parent MRR, the `base+offset` is the starting address of the child MRR.

For example, Fig. 6 is a linked list node structure. We will build a multi-level MRR model like Fig. 7. MRR `node` covers all 22 bytes of this linked list structure. MRR `buffer` addressed by `node` address has the child MRR of `node's` and covers 14 bytes of this structure. MRR `name` addressed by `buffer` address has the level 2's MRR and covers 10 bytes of this structure.

### MRR generation

This section describes how to generate a MRR. Overall, there are two types of MRR in a MRR tree, root MRR



<pre> 1 typedef struct _Node{ 2     char buffer[128]; 3     struct _Node *next; 4     struct _Node *prve; 5 }Node; 6 void fun(){ 7     Node * node = (Node *)malloc \ 8         (sizeof(Node)); 9     node-&gt;next = NULL; 10    node-&gt;prev = NULL; 11    gets(node-&gt;buffer); 12 } </pre>	<pre> 1 &lt;fun&gt;: 2 push rbp 3 mov rbp, rsp 4 sub rsp, 0x10 5 mov edi, 0x98 6 call _malloc 7 mov [rbp+node], rax 8 mov rax, [rbp+node] 9 mov qword ptr [rax+0x88h], 0 10 mov rax, [rbp+node] 11 mov qword ptr [rax+0x90h], 0 12 mov rax, [rbp+node] 13 mov rdi, rax 14 call gets </pre>
--	--

**Fig. 4** The source code and assembly code of the linked list example

and child MRR. We have different generation methods for them. A root MRR describes a memory block allocated by the system. e.g., a memory chunk returned by `malloc`. When the system allocates a block of memory, the starting address and block length are known. We generate a new MRR tree and set the root MRR node according to the information of the newly allocated memory block. The root MRR generation rules also apply to stack memory. We identify the stack subtraction operation (e.g., `sub rsp, 0x20`) near the entry point of each function as the allocation operation of the stack memory block and generate a new MRR tree for stack memory block.

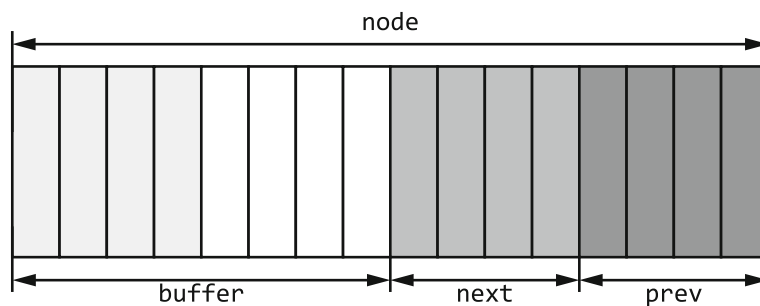
We generate child MRR based on the “base+offset” memory access pattern. Base-pointer addressing (e.g., `newptr=ptr+index`) assigns a new pointer that is the sum of an offset value and a base pointer. This operation reveals fine-grained boundary information of memory objects, because it is frequently used for accessing a member in a data structure. The data structure starts from the address of the base pointer, and the member locates at the offset address in that data structure. Therefore, for all “base+offset” memory accessing patterns, if the base is already the starting address of a MRR *A*, we generate a new child MRR for *A*.

### MRR length

The last section discussed two rules to generate root MRR and child MRR, but we only mentioned getting the starting address from memory allocation or base pointers. We still need to find the ending address so as to get the complete boundary of a memory range. Because we are working on binary code, source code level information such as type and data structure is not available. We have to infer the memory range length from the instructions in program execution.

Our method uses a heuristic to infer the length: the first “write” operation to the memory range. Typically, a memory range is initialized by the first write operation after it is allocated, so the initialization “write” naturally covers the whole memory range and reveals the boundary information. Therefore the length of the first write operation is a good heuristic for inference of the length of the memory range.

We still use the linked list example to explain how to decide the range of a MRR. As shown in Fig. 8, the code initializes the linked list node. The code of line 2 in left hand side of Fig. 8 and line 6 in right hand side generates a root MRR for node. The calculation operation of line 3 in left and line 9 in right generates a child MRR of node



**Fig. 5** Memory ranges in the linked list example

```

1 typedef struct _Buffer {
2     int id;
3     char name[10];
4 } Buffer;
5
6 typedef struct _Node {
7     Buffer buffer;
8     struct _Node * next;
9     struct _Node * prev;
10 } Node;

```

**Fig. 6** A linked list node structure

for next. The memory write operation at line 9 writes 4 bytes, so we set these 4 bytes as the accessible range of pointer next. Similarly, Line 11 applies the same operation. Assuming the id of the node MRR is 1, MRR id of next is 2, MRR id of prev is 3, MRR id of buffer is 4, MRR id of buffer.id is 5, MRR id of buffer.name is 6, the memory identifier map is shown in Fig. 9.

In practice, the first write operation does not always write all memory range inside the boundary. There could be situations when the first write operation writes a larger or smaller memory range. We provide a detailed discussion in Section [First write](#). Our discussion result shows that no matter the first write operation write more or less memory chunks, MAI can still correctly detect the memory access errors.

### MRR state

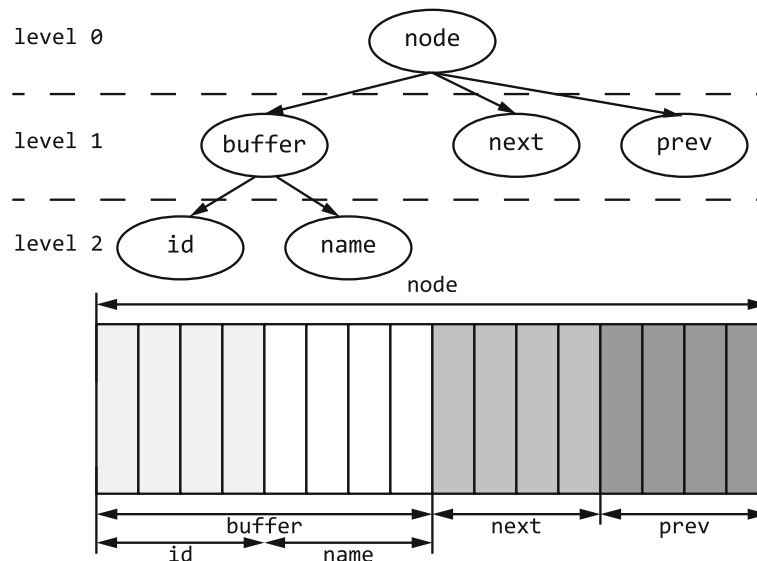
In addition to the starting and ending address, MRR can record the state of the memory range. This is important for policy checking. Our system records three different states in the life cycle of a memory range: *uninit*, *used*, and *free*. When a program allocates a block of memory, the state of the MRR is uninitialized (*uninit*). After some value is written into the memory range, the MRR state will be changed to (*used*). When the operating system free the memory area, the according MRR state is changed to (*free*). Figure 10 shows the memory state transition diagrams. The state is used for detection of memory access errors as shown in “[Error detection](#)” section.

### MRR deletion

After a memory block is freed, no pointer should be able to access this memory; otherwise, it is a use-after-free error. In our multi-level MRR system, we remove the according MRR tree, mark its state as *free*, and also delete the id number. Particularly, the operating system only free a memory block which has been allocated before, so correspondingly, the root MRR and the whole tree are deleted in our system.

### Pointer monitoring

So far we have elaborated the system we designed for record fine-grained memory boundary and state. As shown in “[Design principle](#)” section, MAI policy describes legal accessing relation between a set of pointers and MRRs. Therefore, we also need to monitor the pointers inside the target program.



**Fig. 7** A MRR tree example

```

1 void fun(){
2     Node * node = (Node *)malloc(sizeof(Node));
3     node->next = NULL;
4     node->prev = NULL;
5     node->buffer.id = 1;
6     gets(node->buffer.name);
7 }

```

```

1 <fun>:
2 push rbp
3 mov rbp, rsp
4 subrsp, 0x10
5 mov edi, 22
6 call malloc
7 mov [rbp+node], rax
8 mov rax, [rbp+node]
9 mov qword ptr [rax+14], 0
10 mov rax, [rbp+node]
11 mov qword ptr [rax+18], 0
12 mov rax, [rbp+node]
13 mov qword ptr [rax+0], 1
14 addrax, 4
15 mov rdi, rax
16 call gets

```

**Fig. 8** The source code and assembly code of initializing the linked list node

Since the core rule of MAI is  $(PS, MRR)$ , we only take care of the memory pointers associated with MRR. We get these pointers naturally from the two ways of MRR generation in 5. One is from the program's allocation operations. When the system allocates a chunk of memory, the pointer it returns is the target pointer we want to monitor. Because this operation means the return pointer can access the chunk of memory. Another way is the program's base-pointer addressing operation. In this operation, we will generate a new MRR and the new  $base+offset$  pointer is our monitoring target.

In the process of pointer monitoring, we also monitor the behavior of using the tainted pointer for memory operations. These memory operations include memory reads, memory writes and memory free. Memory read and write operations are performed by using a tainted pointer to read and write memory bytes. The memory free operation is the system's deletion of the memory chunk pointed by the tainted pointer. We insert the MAI integrity check before these operations to ensure the security of the program.

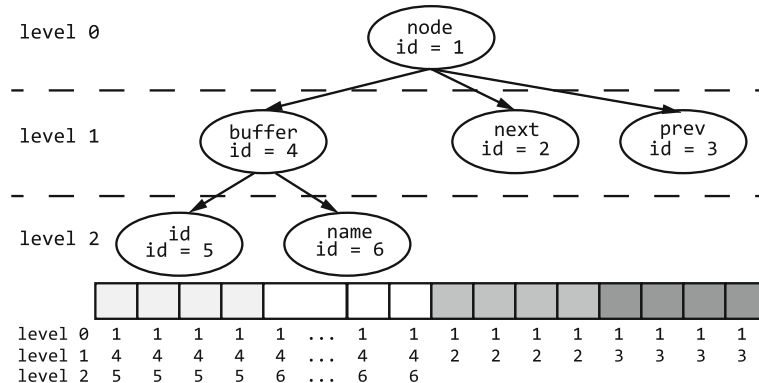
In practice, pointers may have various aliases, e.g., different pointers pointing to the same address. In our system, we tackle the pointer alias problem by dynamic multi-tag taint analysis. We keep track of the propagation of pointers and taint all aliases of a pointer with the same tag.

### Policy

After introducing MRR and pointer monitoring, now we can associate a pointer set with MRRs to build the MAI policy  $(PS, MRR)$ . Every MRR is connected to those pointers that can access the starting address of this MRR. In another words, a rule  $(PS, MRR)$  means the pointers inside PS should be used to access the memory objects inside the range of MRR.

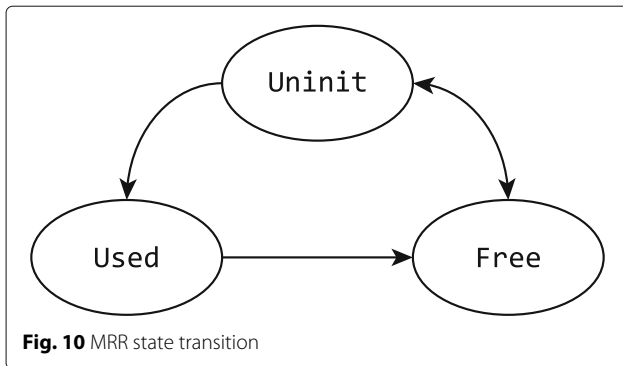
In general, the policy in MAI can be formally interpreted as follows.

**Definition 3** A memory access  $base+offset$  is legal, if and only if:  
if  $base \in PS$ ,  $base+offset$  must be in MRR.



**Fig. 9** The MRR tree and memory range after initialization





Particularly, pointers associated with a MRR is also legal to access all its child MRRs, because all child MRRs are sub ranges. In our multi-level MRR systems, this feature also helps us track more information when an overflow happens, e.g., the offset to the head of the memory chunk, where this chunk is allocated. The information is essential in program analysis and it can save lots of time for analysts.

When a memory out-of-bounds-write occurs, we can know where the vulnerable memory area is allocated in the code segment based on the pointer used, which sub-region of the memory block it belongs to, what is the offset to the first byte of the memory chunk and where is the memory block allocation by its parent node, and so on, etc.

For example, when we detect the name in Fig. 7 which has an overflow vulnerability, we can know that the name memory region is the sub-region of `buffer` and the `buffer` is the sub-region of `node`. So we know where this vulnerable memory is allocated, where vulnerable memory is written, what the boundaries inside this vulnerable, and much other useful information. All this information can be queried by this pointer's MRR chain.

### Error detection

Previous sections present the detailed mechanism of MAI. In this section, we describe how to detect fine-grained memory access errors based on MAI system. In general, we examine whether a pointer violates MAI policy from the following two aspects.

- 1 The pointer is accessing correct location.
- 2 The pointer is performing correct operation on the memory state.

If one of the aspects failed to pass the detection, an alert indicating memory corruption is automatically reported.

By checking different combinations of pointer operations and memory states, MAI is able to detect three memory access errors as shown in Table 1.

**Table 1** Operation type error: the type of error generated by different types of operations accessing different memory states

Type	State	Uninit	Used	Free
read	uninitialized-read	-	-	read use-after-free
write	-	-	-	write use-after-free
deallocation	-	-	-	double-free
allocation	chunk overlap	chunk overlap	-	-

'-' indicates legal access

If the memory access behavior uses a pointer in PS and its address is not inside the corresponding MRR, this is an out-of-bounds error. In this violation case, we could furthermore perform root cause diagnosis, as is shown in Table 2. If the victim memory chunk of the invalid memory access is not of root MRR, we are able to draw the conclusion that this out-of-bounds error happens inside a memory chunk. For example, If an out-of-bounds error happens in the heap and the level of the overflowed memory's MRR is not the root MRR, this error is a fine-grained memory access error.

In addition to the vulnerability types, we also provide some information about Trouble-Shooting. Because we can get the MRR generation address, MRR base address, and parent MRR node by querying the MRR tree. We can get the information about where this memory chunk is first to be written, what the overflowed memory region belong to, where is this memory chunk allocated and so on.

### Implementation

This section presents the implementation details of MAI. We implement MAI as an enforcement tool based on the Valgrind dynamicbinary instrumentation framework (Luk et al. 2005; Hundt et al. 2005).

We reused the translation, IR statement analysis and instrumentation of the intermediate language VEX of the Valgrind framework. The core technique is dynamic instrumentation. We insert vex statements to implement the function of MAI, like pointer monitor, MRR maintaining. The whole implementation includes about 4,000 lines of C code in total.

**Pointer monitoring.** The pointer monitoring function is implemented based on taint analysis. We use a dynamically expanded shadow memory to taint the pointer store

**Table 2** Four types of pointer overflow error according to overflow location

-	Heap	Stack
inside	intra-heap overflow	intra-frame overflow
outside	inter-heap overflow	out-frame overflow

address with MRR id. Shadow memory (Nethercote and Seward 2007a) consists of shadow bytes that map to individual bits or one or more bytes in main memory. In order to save space, the size of the shadow memory is dynamically expanded. When the pointers are propagated, we copy the id to another. Unlike taint analysis, our implementation uses fewer but sufficient rules which is mentioned in Section [Pointer monitoring](#). This implementation allows us to circumvent many traditional problems of taint analysis, like overtainting and undertainting, because we greatly reduce the instruction complexity of tracking taint.

**MRR record.** We record the MMR id and memory state information on another dynamically expanded shadow memory, in order to ensure that the pointer monitoring shadow memory will not disturb the content of the MMR's. Other MRR information is recorded in a global variable as a tree structure, which can be queried by MMR id. We allocate 16-bits shadow memory for each byte. The high 14-bits used to record memory privilege identifier and the low 2-bit used to record memory state.

**Policy.** The method we build MAI policy (PS,MRR) is implemented by id field. Pointers to be monitored are tainted with an id and MRR has an id field. We bind the pointers and MRRs with the same id, which means the pointers should be used to access the memory objects inside the range of MRR with same id.

**Error detection.** We insert MAI policy detection before five types of operation: base-pointer addressing operation, read operation, write operation, allocation operation and deallocation operation. Before these operations, we examine if the id of pointer and target address's MRR are equal and if the memory state match the operation type.

#### First write

As mentioned earlier, the method we use to determine the MRR length encounters two problems, the length of the first write is less than the actual length of MRR and the length of the first write is longer than the actual length.

In the situation that the length of the first write is less than the actual length of MRR. The MAI will automatically expand the range of this MRR, when the remaining range of this MRR is written for the first time. The first write operation is longer than the actual length is undoubtedly a malicious operation and may break the program. we will describe how we detect this situation.

There are two situations to discuss here. We will use the structure shown in the Fig. 11 as an example. In this example, the program will write 6 bytes to the array a which is an overflow operation. In the first situation, the switch variable of the program has been written before the overflow operation. Shown in the Fig. 12-a, the switch has already been set a MRR(id=2). When program wants to use array a's MRR(id=3) to access switch's MRR(id=2), we will detect this overflow error. In the second situation,

```
1 struct P {  
2     int num;  
3     char a[5];  
4     int switch;  
5 }
```

**Fig. 11** An data structure showing first write

the switch variable of the program has not been written before the overflow operation. Shown in the Fig. 12-b, the switch has no MRR. In this case, we will temporarily recognize the overflow operation as a normal operation and sets a MRR(id=3) whose size is larger than it should be. Since the switch has not been written to the data, this setting will not cause malicious consequences. When the program reads or writes the variable switch for the first time, this error can be detected because MAI finds that the memory already has a MRR. This is a kind of delay detection.

#### Evaluation

In this section, we present the evaluation scheme and result of MAI. Basically, we design experiments to answer the following three research questions.

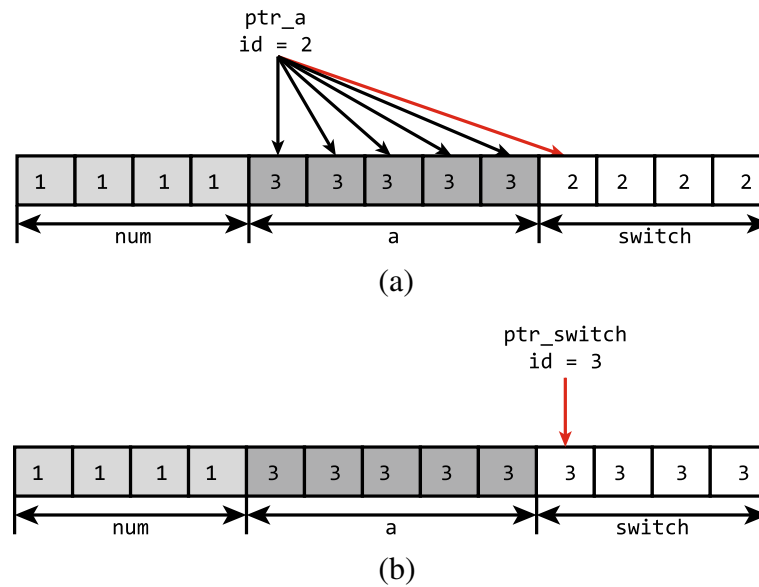
- 1 **RQ1:** Is MAI able to detect various memory access errors?
- 2 **RQ2:** How many false positives and false negatives can MAI produce?
- 3 **RQ3:** What is the overhead of MAI?

As the response to RQ1, we first apply MAI to a CTF problem set including 10 examples of different types of memory access errors. To check MAI's effectiveness in practice, we run MAI and Valgrind on 10 real-world programs and compare the result. We also elaborate a case study to give a detailed answer to RQ1. As the answer to RQ2, we apply MAI to 50 randomly selected CTF programs and check the result of false positives and false negatives. In response to RQ3, we compare MAI's performance with Valgrind.

The evaluation runs on a 64-bit Ubuntu 16.04 desktop and 4GB RAM. MAI is compiled by GCC 5.4.0. The Valgrind version is 3.14.

#### CTF challenges

To evaluate that MAI is able to detect various types of memory access errors, we collected 10 difficult programs



**Fig. 12** Different cases of first write

(weight score  $\geq 25$ ) from popular CTF events in the past two years. Weight score is an per-event value, depends on tasks and organization level, participated teams or previous years weight used. Events with high weight score means that they are widely known and their competition programs have a high quality.

Table 3 present the evaluation result. The first and second column shows the name of challenges and the event name. The third column shows the type of memory access error in that challenge. The last two columns present MAI's detection result and the error type.

**Table 3** The detection result of MAI on CTF challenges with ten different memory corruption bugs

Challenges Name	Event	Bug Type	Detection
babyheap	0ctf2017	use-after-free	✓
aur	casw2017	double free	✓
house_of_c4rd	0ctf2018quals	integer overflow	✓
scv	casw2017	stack overflow	✓
simple_memo_pad	codebluectf2017	intra data-structure overflow	✓
1000levels	hitbctf2017	read uninit	✓
woO2-fixed	TUCTF 2016	abuse global variable	✓
fastbin	0ctf2017	chunk overlap	✓
babyheap	Codegate 2018	heap write overflow	✓
babyheap	SECUINSIDE 2017	heap read overflow	✓

The evaluation result shows that MAI successfully detects all memory access errors and reports the correct error type in all test programs. Particularly, the “abuse global variable” error type from the challenge “woO2-fixed” is an out-of-bounds access caused by a dangling pointer, which points to a memory block that has been previously freed but reallocated. Since the newly allocated memory has a different id than the pointer, MAI successfully captures this memory access error.

### Real world programs

In order to evaluate MAI's capability in practice, we apply MAI and Valgrind to 10 real-world programs with known memory access errors and compare the detection result. Openjpeg and Jasper are Image toolkit, OpenSSL is a toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. Vlc is a video player. Unrar is a compression software. Iselect is an interactive line selection tool for textual files and Polymorph is a filesystem. libxaac is an audio decoder library. Mutt is a command-line email client. Iselect and Polymorph are obtained from the BugBench suite (Lu et al. 2005). The rest eight programs are obtained from Exploit-db and cvedetails.com.

In this experiment, we run MAI with a fuzzing tool (Godefroid et al. 2012; Sutton et al. 2007) and it detects three zero-day intra-frame overflow error in the test program libxaac. We have reported the bugs and the security team has conducted a severity assessment on this issue. Based on their published severity assessment matrix they were rated as critical severity. CVE numbers will be assigned to these vulnerabilities once they finish developing an update.

We present the detailed description of the experiment as follows. AFL mutates the provided seed corpus and then feed them to the target program libxaac and MAI runs libxaac to accept the input. MAI detection tool detects if the input causes memory access errors. If it has, we will manually analyze this bug based on the error report. At the beginning of the program, it will allocate a large memory (64MB) to store its data structures. It means the program store many structures in a memory chunk. If one of these structures can be overflowed, other structures will also be affected and this overflow does not cross the boundary of this large memory chunk. Thus this overflow is an intra-frame heap overflow. We have fuzzed three data structures with this vulnerability.

Table 4 lists the program name, CVE numbers and the detection ability of MAI and Valgrind. MAI with full checking was able to detect and prevent all of the errors. The bugs of program *iselect*, *polymorph*, *libxaac* and *netatalk* is intra-frame overflow. An attacker will first overflow some intra-frame local variables, then overflow the out-frame variable to hijack control flow. MAI can detect this overflow when an attacker overflows the first intra-frame local variable, but Valgrind memcheck detects this overflow when the attacker overflows the out-frame variable. We will explain *Netatalk* with intra-frame overflow as an example.

### A case study

Netatalk is an open source file sharing server. An UNIX, Linux or BSD system running Netatalk is capable of serving many Macintosh clients simultaneously as an Apple-Share file server (AFP).

The vulnerability in Netatalk 3.x is shown as the following code 13. Missing length check of the arguments in memcpy is the root cause.

In this function, `dsi->commands` is under the control of the attacker. The variable `dsi->commands` is a

char array and the size parameter is limited to a maximum value of 255. So an attacker can write up to 255 bytes starting from the address of the `dsi->attn_quantum` which is a 4-byte integer.

`attn_quantum` is a member of the struct DSI which is allocated in a 10096 bytes chunk in the heap. The part of this struct is shown as code 14. Because of only 251 bytes we can overflow, the members can be controlled are `datasize`, `server_quantum`, `serverID`, `clientID`, the `commands` pointer, and partially into data. The overflow bytes cannot write over the boundary of this heap chunk.

Luckily the life of the `commands` pointer begins shortly after a new connection is forked to its own process. `commands` passes through the system based on a global jump table pointer defined in `etc/afpd/switch.c` called `afp_switch`, so the exploit of this vulnerability is feasible.

The initialization of this struct is shown as code 15. In line 4, the program allocates 10096 bytes in heap. In MAI, we generate a new MRR (LV0) for this heap and taint the point `dsi` with this new MRR id.

In line 7-9, the program initializes `attn_quantum`, `server_quantum` and `dsireadbuf`. According to the offset to the pointer `dsi`, we generate three new MRRs (LV1) and set these MRRs as the child node of the pointer `dsi`'s and taint their own pointers.

As mentioned in Section [Memory range record](#), we taint the shadow memory bytes according to the bytes first written by the pointer. This processing will set the range pointer can access. So when the function `memcpy` wants to overflow the member `dsi->server_quantum` with the pointer `dsi->attn_quantum`, MAI will detect this error.

Valgrind sets memory boundaries based on the size of heap chunks, but this overflow operation happens inside a heap chunk and does not over the chunk boundary. Therefore, Valgrind cannot detect this overflow vulnerability.

### False positives and false negatives

We evaluate the false positives and false negatives by testing 50 CTF programs from 25 CTF events whose weight are higher than 25. We randomly select programs which have memory access error vulnerability in these CTF events to ensure the randomness of the testing. The result shows that the false positive rate is 4% and false negative rate is also 4%.

The main cause of false positives is the special operation of the program itself. Assuming a c-type char array is allocated, the program writes a zero at the max length of this array to identify the end of the string. In our MRR's generation rule, we will generate a subordinate level MRR (with offset max length) for this byte which is the last byte of the array. When the program writes this array from the beginning, we will generate another subordinate level

**Table 4** The comparison detection result on 10 real programs

Program Name	CVE	Error Type	Detection	
			Valgrind	MAI
openjpeg	2016-9572	coarse-grained	✓	✓
jasper	2016-9583	coarse-grained	✓	✓
openssl	2016-7054	coarse-grained	✓	✓
vlc	2017-8311	coarse-grained	✓	✓
unrar	2012-6706	coarse-grained	✓	✓
mutt	2007-2683	coarse-grained	✓	✓
iselect	N/A	fine-grained	×	✓
polymorph	N/A	fine-grained	×	✓
libxaac	N/A	fine-grained	×	✓
Netatalk	2018-1160	fine-grained	×	✓

```
1 /* parse options */
2 while (i < dsi->cmdlen) {
3     switch (dsi->commands[i++]) {
4         case DSIOPT_ATTNTQUANT:
5             memcpy(&dsi->attn_quantum, dsi->commands + i + 1, dsi->commands[i]);
6             dsi->attn_quantum = ntohl(dsi->attn_quantum);
7
8         case DSIOPT_SERVQUANT: /* just ignore these */
9             default:
10                 i += dsi->commands[i] + 1; /*forward past length tag + length */
11                 break;
12     }
13 }
```

**Fig. 13** Netatalk vulnerable code

MRR(with offset 0). So when the MRR(with offset 0) to write the MRR(with offset max length), MAI will report an OOB error which is a false positive. Another example is using different operations to write an int array. When the array is initialized, the program uses the following code in Fig. 16.

For each int variable, we give it an MRR that can access 8 bytes, but this array is used later. The source code is shown in Fig. 17.

In our MRR's detection rule, line 3 is an OOB error which is a normal operation. The above two cases have not appeared in other test cases, so these two cases may rarely appear in other programs. These two types of false positives are also well investigated according to the context of error reporting and error reporting points.

There is also other special operation, for example, some functions in glibc, such as *malloc*, *free*. They will use a pointer to over-access another memory chunk to do some safety checks. These operations will lead to false positives.

Our solution is to stop memory bounds detection when the program runs the code in libc which will probably raise false positives.

False negatives are mainly caused by the memory access errors occurring at the locations where MAI does not monitor, such as a global variable in .data segment or function pointer in the .got segment. It is because the generation of the root MRR is based on the program allocation operation. But in the case of the above segment, it has been allocated before the program is running. So we cannot generate an MRR for this place and cannot detect out of bounds accesses in these places. In addition, since the use of memory on these segments usually directly show as a memory address rather than a form of *base\_point+offset*, it is difficult for MAI to tell if it is a pointer or just an integer value. We provide a manual extension to mark memory pointers and memory bytes. Users can track and do boundary detection on any pointers,

```
1 #define DSI_DATASIZ 65536
2 typedef struct DSI {
3     struct DSI *next; /* multiple listening addresses */
4     AFPObj *AFPObj;
5     int statuslen;
6     char status[1400];
7     char *signature;
8     struct dsi_block header;
9     struct sockaddr_storage server, client;
10    struct itimerval timer;
11    int tickle; /* tickle count */
12    int in_write;
13    int msg_request; /* pending message to the client */
14    int down_request; /* pending SIGUSR1 down in 5 mn */
15    uint32_t attn_quantum;
16    uint32_t datasize;
17    uint32_t server_quantum;
18    uint16_t serverID, clientID;
19    uint8_t *commands; /* DSI recieve buffer */
20    uint8_t data[DSI_DATASIZ]; /* DSI reply buffer */
21    size_t datalen, cmdlen;
22    ...
23 }
```

**Fig. 14** DSI struct



```

1 DSI *dsi_init(AFPObj *obj, const char *hostname, const char *address, const char *port)
2 {
3     DSI *dsi;
4     if ((dsi = (DSI *)calloc(1, sizeof(DSI))) == NULL)
5         return NULL;
6
7     dsi->attn_quantum = DSI_DEFQUANT;
8     dsi->server_quantum = obj->options.server_quantum;
9     dsi->dsireadbuf = obj->options.dsireadbuf;
10
11     /* currently the only transport protocol that exists for dsi */
12     if (dsi_tcp_init(dsi, hostname, address, port) != 0) {
13         free(dsi);
14         dsi = NULL;
15     }
16     return dsi;
17 }

```

**Fig. 15** The initialization code in Netatalk

including global variables and function pointers in the .got segment.

### Performance

In this section, we evaluate the runtime overhead of 10 real programs. The average overhead of MAI is around 2-26 times (Fig. 18). The black part is MAI's overhead, and the gray part shows the overhead of Valgrind. In most cases, MAI's overhead is a little higher (less than twice) than Valgrind's overhead, and there are also some cases that MAI is lower than Valgrind. The reason is that when we do the optimization, we remove some of Valgrind's own functions and instruments, such as the execution state of the statement, and greatly optimize the memory read and write of our own system.

There are two primary sources of overhead in MAI. One of the sources is that Valgrind still has the operations that are not necessary for the implementation of MAI. Another source is the runtime overhead of metadata accesses. Frequent reads and writes to shadow memory are required in the generation of new MRR and detection of MRR's queries. In Fig. 18, challenges like *scv*, *Houseofcard*, *Auir* and programs like *dnstrace*, *ncompress* have few MRRs, so the overhead is low. However, programs like *netatalk* and *libxaac* generate a large number of MRRs due to frequent heap memory allocation operations and complex data structure.

```

1 int array[10];
2 for(int i = 0 ; i < 10 ;
3 i++){
4     array[i] = 0;
5 }

```

**Fig. 16** The first method of initializing arrays

We also observe that larger number of MRRs leads to higher cost. In our design, we generate a new MRR for each member variable of each data structure to achieve the effect of fine-grained detection. Reuse of a member variable does not generate a new MRR. Therefore, the number of MRRs in the MAI is only related to the number of data structures owned by the program itself. The number of data structures in a program is limited, so the cost is acceptable.

### Related work

In this section, we introduce the background of this research and the line of work closely related to our work. The research areas mostly close to ours are exploit mitigations and data-structure recovery.

#### Exploit mitigations

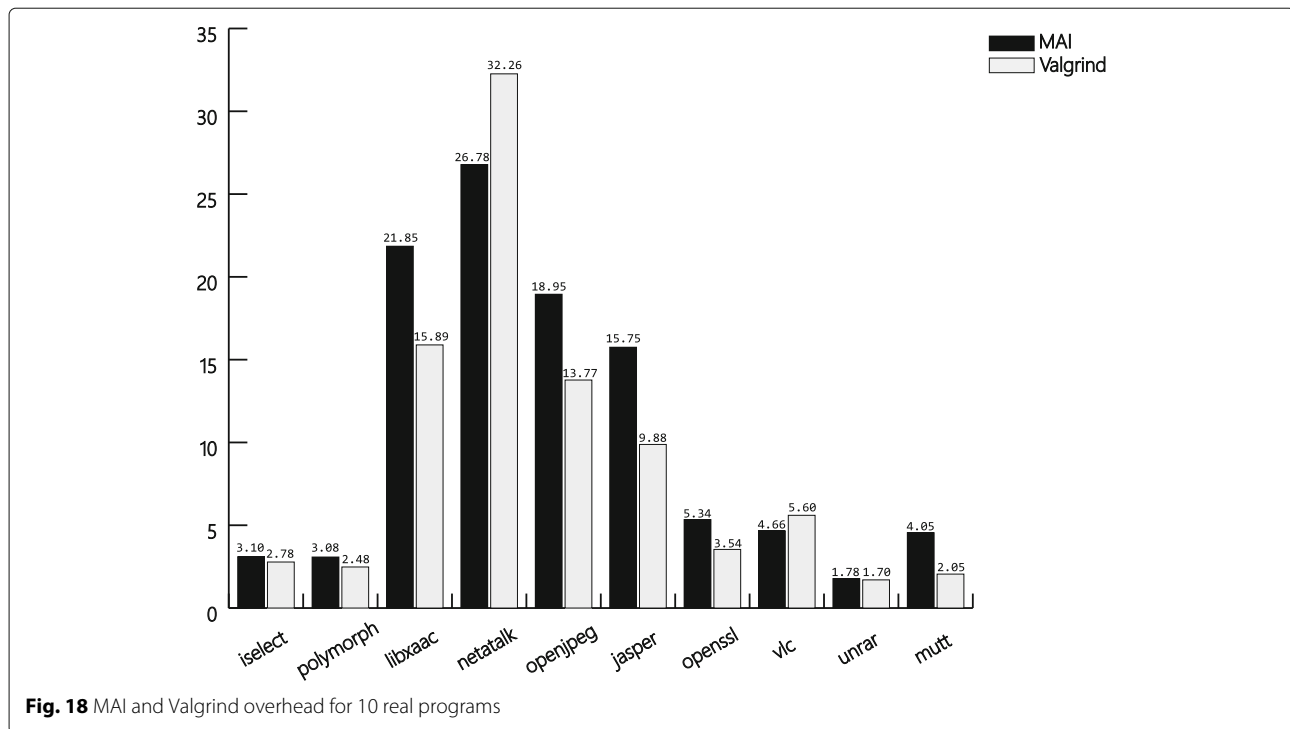
Control flow integrity (Abadi et al. 2005; Kuznetsov et al. 2014) leverages static analysis and program instrumentation to guarantee the runtime control flow follows the precomputed correct control flow graph, and thus could detect and prevent an important exploit vector: control flow hijack, however, CFI only detects exploit attempt when the control flow deviates from the legit control flow graph, if the exploitation does not involve control flow violation (e.g., Data-Oriented Programming (Hu et al. 2016)), CFI is not able to detect the exploit. Data flow integrity is another technique which prevents invalid read and writes operation by calculating a valid data flow graph at compile time. Object Type Integrity (Burow et al. 2018) is

```

1 char * p = &(array[3]);
2 ...
3 p++ = value;

```

**Fig. 17** The second method of initializing arrays



designed to protect the programs written in C++ as an orthogonal policy as CFI. OTI tracks the assigned type for every object at runtime. When the object's type is used for dynamic dispatch, OTI can verify that the type is uncorrupted. OTI get object information while a program is compiled.

#### Memory corruption detection technique

Traditionally, dynamic vulnerability analysis technique relies on explicit signals such as software crash or system panic to determine whether a sink point is reached. However, this result in a lot of false negatives because an input which triggers a memory corruption vulnerability does not necessarily crash the target program. To facilitate detecting trigger of vulnerabilities and analyzing root cause, a majority of research works focuses on design corruption site sensitive vulnerability detection technique to identify occurrence of memory corruption as early as possible. Another important area of research is vulnerability detection technique designed to find vulnerability at the site corresponding to the root cause of the vulnerability. These works are closely collaborate with vulnerability discovery technique (e.g., fuzzing) to generate sink point closer to the memory corruption site.

Based on dependency on source code information, the vulnerability detection technique can be categorized into two families, one relies on source code information while the other does not. AddressSanitizer (or ASan) (Serebryany et al. 2012) is an open source compiler extension

developed by Google that detects memory corruption bugs such as buffer overflows or dangling pointer accesses (e.g., use-after-free). AddressSanitizer is based on compiler instrumentation and directly-mapped shadow memory. Asan detects memory overflow error by inserting a special memory between two adjacent memory blocks during program compilation. Wookhyun et al. (Han et al. 2018) enforces two extra properties and achieve better memory corruption detection performance. Although the above approaches incurs a relatively low runtime overhead, it also changes the memory layout, making it tricky to detect some special vulnerabilities. Of course, the dependency on source code is a major difference between address sanitizer and our work.

Another research direction is designing and developing binary analysis tools which are able to perform vulnerability detection without source code. The lack of type information brings a lot of challenge to vulnerability detection. Valgrind is an instrumentation framework for building dynamic analysis tools. Its core memory error detector—memcheck—detects memory-management problems, primarily for detecting memory corruption for program compiled from C and C++ language. When a program is executed under memcheck's supervision, all memory read and write operations are checked, and calls to malloc/new/free/delete are intercepted.

Valgrind uses runtime information to recover the stack frame information and maintain the basic heap layout

information, however, as is shown in “[Real world programs](#)” section, such coarse-grained type boundary information is not enough to detect some intra memory chunk corruption.

### Data structure recovery

In the absence of type information (e.g., debug info, symbol-table), approximating and inferring data structure is a very challenging research topic. Gogul addresses the problem with value-set analysis (VSA) in (Balakrishnan and Reps 2005), they show the effectiveness by identifying arrays, local variables, and heap-allocated data structures.

Asia (Slowinska et al. 2010) uses a dynamic approach to extract data-structure. These works are similar to ours in the sense that we are both able to extracting information related to the data structure and its sub-fields. However, these works do not provide us the ability to decide the privilege of instructions tied to memory chunks, including its subfields.

Traditional data structure recovery is different from memory bounds recovery techniques for memory error detection. Traditional data structure recovery techniques focus on restoring all data structures as accurate as possible. But for memory bounds recovery techniques in the field of memory error detection, we only need to recover the memory bounds corresponding to pointers used by the program during the running of a process. VSA (Balakrishnan and Reps 2005) shows the effectiveness by identifying arrays, local variables, and heap-allocated data structures. Asia (Slowinska et al. 2010) uses a dynamic approach to extract data-structure. The accuracy of these techniques is around 80%-90%. However, these data structure recovery techniques do not consider time sequence of memory access which is important for memory error detection. For example, DDE uses the spacing between different offsets to decide the range of a pointer. This method cannot provide the information whether a block of memory is written or not. So it is hard to detect some temporal safety memory error like *uninitialized-read* and *use-after-free*. Moreover, it is not enough to recover the data structure. We also need the relationship between pointers and memory used during the running of the program. Additionally, it is necessary to provide information pertaining to the root cause of the vulnerability with more details. Traditional data structure recovery techniques cannot provide this information. In this paper, we use a new technique to get memory bounds information mentioned in Section [Memory range record](#).

### Conclusion

In this paper, we focus on an interesting and challenging research problem: detect fine-grained memory access errors in binary code. While plenty of research works have been proposed to explore memory corruption detection

with the help of source code and type information or some basic runtime memory usage information, they are barely useful to recover fine-grained memory boundary information for binary executables and thus fail in some memory corruption detection. We propose Memory Access Integrity, an effective method to infer and check the memory access policy between pointers and memory blocks. We implement a prototype system that facilitates memory access error detection for off-the-shelf binaries and prove the effectiveness of our method.

We demonstrated the utility of MAI with various categories of memory corruption bugs, MAI is able to detect all of the evaluated memory corruption bugs and facilitate root cause diagnosis. MAI's ability to recover fine-grained memory boundary significantly improve the detectability of memory corruption bugs happening in a single stack frame or a single heap chunk. We plan to release the source of our current implementations as well as migrate the implementation to Valgrind platform. In addition, we will explore to find the theoretic upper bound of memory boundary information that could be extracted from binaries.

### Discussion

In this section, we discuss three limitations of MAI, which lead to false positives and false negatives. Then we discuss the solutions taken in two special cases and the risks that may arise.

**Compiler optimization.** The compiler optimizes the program when it compiles the source code into binary. To minimize the time taken to execute a program, the compiler uses optimization strategies like peephole optimizations, local optimizations, loop optimizations and so on.

MAI method is based on the “first write” and “base+offset” addressing operations. the “first write” operation in MAI's design decides the length of an MRR. It means the length of the memory region that can be used in a memory object. The length also reflects the “valid” area of the memory object. The compiler's optimization operation may reduce the number of variables and increase or decrease the initialization length of the variables, but there is no case like initializing 10 bytes but using 15 bytes. So the change of “first write” operation do not cause false positives of false negatives.

However, compiler optimization that affects “base+offset” addressing operations affect the implementation of MAI. This kind of optimization strategy makes the memory model we restore from binary different from the original data structures, but as long as the read and write behavior is consistently under the unified standard, it will not affect the MAI detection results. For example, there is an optimization that replaces all offsets of “base+offset” addressing operation with the offset

between the memory objects and page start address. In this situation, the MAI method generates a root MRR node at the start address of each page and restore the sub-regions inside each page. This memory model is different from the original data structures but does not affect the MAI detection process.

If optimization strategy make the “base+offset” addressing operation of the same memory object inconsistent, this kind of strategy leads to false positives. For example, compiler may merge some consecutive pointer offset operations, like merging “ $p2 = p1 + \text{offset1}$ ”  $p3 = p2 + \text{offset}$ ” to “ $p3 = p1 + \text{offset}$ ”. After this optimized operation, MAI method generates a child MRR node of  $p1$ ’s MRR node for  $p3$ , but actually  $p3$ ’s MRR should be the child MRR node of  $p2$ ’s. If and only if program uses pointer  $p2$  to access  $p3$ ’s MRR range, MAI method reports a false positive error.

**Union data structure.** The union data structure causes false positives in the MAI method. This is a limitation. In future work, we will use union data structure identification techniques to solve this problem. Fortunately, the union data structure is not common in programs.

**Global variables.** MAI method can check heap and stack objects access. However, due to the lack of allocation operation before the global variable is used, the MAI method cannot achieve automatic monitoring of global variables. We try to identify global variables by static analysis, but this method produces many false positives and false negatives. In future work, we will use other global variables identification techniques to solve this problem.

**Delay detection.** Mentioned in subsection [First write](#), we use “delay detection” solution to detect “malicious first write” error. This technique can stop exploit before it causes harm, but it still causes data pollution. In addition, there is a problem with the location of the vulnerability in the error report, because it is difficult for MAI to determine whether the location of the vulnerability is the previous overflow write operations or an out-of-bounds access by the new pointer.

**Redundant MRR node.** When the MAI method handles some operations, it generates redundant MRR nodes, which does not affect the detection of MAI but generates extra time overhead. There are two cases to discuss here. The first case is to use a loop to write memory continuously. Each round of loops contains a pointer addressing operation and a write operation. According to the rule of MRR’s generation, We generate a child node MRR for each round. We refer to these child nodes as redundant nodes. Because the MRR of the pointer which points to the first address of the memory is the parent node of these child nodes, and its range is the sum of these child nodes. This is in line with the actual situation and does not cause false positives and false negatives. In order to reduce the time overhead of these redundant nodes’s generation, we merge

loop-writing operations by identifying the characteristics of such operations.

The second case is that programs use functions such as “memset” to initialize the entire block of memory. In this case, the MAI method generates a child node with the same range as the root node and the subsequently generated child nodes will become the child node of this node. This node is a redundant MRR node and does not affect the detection of MAI. In order to reduce the time overhead, we hook the libc functions like “memset” and ignore their execution when their parameters meet the conditions for generating redundant nodes.

#### Acknowledgements

Not applicable.

#### Funding

Not applicable.

#### Availability of data and materials

All public dataset sources are as described in the paper.

#### Authors’ contributions

WL, DX and XG designed the study. WL, XX and Fg performed the experiments. DX, WL and WW wrote the paper. XG, YW and QZ reviewed and edited the manuscript. All authors read and approved the manuscript.

#### Competing interests

The authors declare that they have no competing interests.

#### Publisher’s Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

#### Author details

<sup>1</sup>School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. <sup>2</sup>University of New Hampshire, Beijing, China. <sup>3</sup>Key Laboratory of Network Assessment Technology, CAS, Beijing, China. <sup>4</sup>Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China. <sup>5</sup>Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.

Received: 16 April 2019 Accepted: 5 May 2019

Published online: 07 June 2019

#### References

- Abadi M, Budiu M, Erlingsson U, Ligatti J (2005) Control-flow integrity. In: *ACM Conference on Computer & Communications Security*. [http://xueshu.baidu.com/s?wd=paperuri%3A%28f6b7e0d5098513f897e156e75fa04af2%29&filter=sc\\_long\\_sign&sc\\_kn\\_para=q%3DControl-flow%20integrity&sc\\_us=1548336401933715558&tn=SE\\_baiduxueshu\\_c1gjeupa&ie=utf-8](http://xueshu.baidu.com/s?wd=paperuri%3A%28f6b7e0d5098513f897e156e75fa04af2%29&filter=sc_long_sign&sc_kn_para=q%3DControl-flow%20integrity&sc_us=1548336401933715558&tn=SE_baiduxueshu_c1gjeupa&ie=utf-8)
- Akriditis P, Cadar C, Raiciu C, Costa M, Castro M (2008) Preventing memory error exploits with WIT. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, pp 263–277. <https://ieeexplore.ieee.org/abstract/document/4531158/>
- Austin TM, Breach SE, Sohi GS (1994) Efficient detection of all pointer and array access errors (Vol. 29, No. 6, pp. 290–301). *ACM*. <https://dl.acm.org/citation.cfm?id=178446>
- Balakrishnan G, Reps T (2005) Recovery of variables and heap structure in x86 executables. Technical Report 1533, Computer Sciences Department
- Bruening D, Zhao Q (2011) Practical memory checking with dr. memory. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. pp 213–223. IEEE Computer Society <https://dl.acm.org/citation.cfm?id=2190067>
- Buchanan E, Roemer R, Shacham H, Savage S (2008) When good instructions go bad: Generalizing return-oriented programming to RISC. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, pp 27–38. <https://dl.acm.org/citation.cfm?id=1455776>

- Burow N, McKee D, Carr SA, Payer M (2018) Cfixx: Object type integrity for c++ virtual dispatch. In: Prof. of ISOC Network & Distributed System Security Symposium (NDSS). <https://hexhive.epfl.ch/publications/files/18NDSS.pdf>.
- Castro M, Costa M, Harris T (2006) Securing software by enforcing data-flow integrity. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. USENIX Association. pp 147–160. <https://dl.acm.org/citation.cfm?id=1298470>
- Checkoway S, Davi L, Dmitrienko A, Sadeghi A-R, Shacham H, Winandy M (2010) Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. ACM. pp 559–572. <https://dl.acm.org/citation.cfm?id=1866370>
- Chen S, Xu J, Nakka N, Kalbarczyk Z, Iyer RK (2005) Defeating memory corruption attacks via pointer taintedness detection. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). IEEE. pp 378–387. <https://ieeexplore.ieee.org/abstract/document/1467812/>
- Condit J, Harren M, Anderson Z, Gay D, Necula GC (2007) Dependent types for low-level programming. In: European Symposium on Programming. Springer. pp 520–535. Technical Report EECs-2006-129, UC Berkeley; 2006
- Dhurjati D, Adve V (2006) Backwards-compatible array bounds checking for c with very low overhead. In: Proceedings of the 28th International Conference on Software Engineering. ACM. pp 162–171. <https://dl.acm.org/citation.cfm?id=1134309>
- Dhurjati D, Kowshik S, Adve V (2006) Safecode:enforcing alias analysis for weakly typed languages. *Acn Sigplan Not* 41(6):144–157
- Godefroid P, Levin MY, Molnar D (2012) Sage: whitebox fuzzing for security testing. *Commun ACM* 55(3):40–44
- Han W, Joe B, Lee B, Song C, Shin I (2018) Enhancing memory error detection for large-scale applications and fuzz testing. In: Network and Distributed System Security Symposium (NDSS). <https://liffeasageek.github.io/papers/han-meds.pdf>
- Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z (2016) Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE. pp 969–986. <https://ieeexplore.ieee.org/abstract/document/7546545/>
- Hundt R, Ramasamy V, Gouriou E, Babcock DJ, Lofgren TC, Rivera JG, Krishnaswamy U (2005) Dynamic instrumentation of an executable program by means of causing a breakpoint at the entry point of a function and providing instrumentation code. US Patent 6,918,110. Hewlett-Packard Development Co LP, assignee United States patent US 6,918,110 <https://patents.google.com/patent/US6918110B2/en>
- Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang Y (2002) Cyclone: A safe dialect of C. In: USENIX Annual Technical Conference, General Track. pp 275–288. [https://www.usenix.org/event/usenix02/full\\_papers/jim/jim\\_html](https://www.usenix.org/event/usenix02/full_papers/jim/jim_html)
- Kuznetsov V, Szekeres L, Payer M, Candea G, Sekar R, Song D (2014) Code-pointer integrity. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp 147–163. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov>
- Lam L-c, Chiueh T-c (2005) Checking array bound violation using segmentation hardware. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). IEEE. 388–397. <https://ieeexplore.ieee.org/abstract/document/1467813/>
- Lattner C, Adve V (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. IEEE Computer Society. p 75. <https://dl.acm.org/citation.cfm?id=977673>
- Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y (2005) Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools, (vol. 5). <http://mir.cs.illinois.edu/~marinov/sp05-cs598dm/ShanLu.pdf>
- Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: *Acn sigplan notices* (Vol. 40, No. 6, pp. 190–200). ACM. <https://dl.acm.org/citation.cfm?id=1065034>
- Maklakov L (2017) Have the Error Code ATTEMPTED EXECUTE OF NO EXECUTE MEMORY. [https://answers.microsoft.com/en-us/windows/forum/windows\\_10-performance/have-the-error-code-attempted-execute-of-no/0a35ae12-2c06-4053-8de8-6492d37a290b](https://answers.microsoft.com/en-us/windows/forum/windows_10-performance/have-the-error-code-attempted-execute-of-no/0a35ae12-2c06-4053-8de8-6492d37a290b)
- Nagarakatte S, Zhao J, Martin MM, Zdanczewicz S (2009) Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Not* 44(6):245–258
- Necula GC, Condit J, Harren M, McPeak S, Weimer W (2005) Ccured: Type-safe retrofitting of legacy software. *ACM Trans Program Lang Syst (TOPLAS)* 27(3):477–526
- Nethercote N, Seward J (2007a) How to shadow every byte of memory used by a program. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. ACM. pp 65–74. <https://dl.acm.org/citation.cfm?id=1254820>
- Nethercote N, Seward J (2007b) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: *ACM Sigplan Notices*, (Vol. 42, No. 6, pp. 89–100). ACM. <https://dl.acm.org/citation.cfm?id=1250746>
- Oleksenko O, Kuvaikkii D, Bhatotia P, Felber P, Fetzer C (2017) Intel mpx explained: An empirical study of intel mpx and software-based bounds checking approaches. <https://arxiv.org/abs/1702.00719>
- Patil H, Fischer C (1997) Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw Pract Exper* 27(1):87–110
- Roemer R, Buchanan E, Shacham H, Savage S (2012) Return-oriented programming: Systems, languages, and applications. *ACM Trans Inf Syst Secur (TISSEC)* 15(1):2
- Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) Addresssanitizer: A fast address sanity checker. In: Presented as part of the 2012 USENIX Annual Technical Conference (USENIXATC 12). pp 309–318. <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>
- Slowinska A, Stancescu T, Bos H (2010) Dde: dynamic data structure excavation. In: *ApSys*. ACM. pp 13–18. <http://www.syssec-project.eu/m/page-media/3/dde-apsys10.pdf>
- Sutton M, Greene A, Amini P (2007) Fuzzing: Brute Force Vulnerability Discovery. Pearson Education. <https://books.google.com/books?hl=en&lr=&id=DPAwwn7QDy8C&oi=fnd&pg=PT4&dq=Fuzzing:+Brute+Force+Vulnerability+Discovery&ots=4xwaG1eHqj&sig=GXXE617bUn6P6DvVjVRGupUeY>
- Xu W, DuVarney DC, Sekar R (2004) An efficient and backwards-compatible transformation to ensure memory safety of c programs. *ACM SIGSOFT Softw Eng Notes* 29(6):117–126
- Yong SH, Horwitz S (2003) Protecting c programs from attacks via invalid pointer dereferences. In: *ACM SIGSOFT Software Engineering Notes* (Vol. 28, No. 5, pp. 307–316). ACM, <https://dl.acm.org/citation.cfm?id=940113>

**Submit your manuscript to a SpringerOpen<sup>®</sup> journal and benefit from:**

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

---

Submit your next manuscript at ► [springeropen.com](https://www.springeropen.com)