AddressWatcher: Automatic Memory Leak Fixing

by

Aniruddhan Murali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Automatic bug fixing has become a promising direction over manual fixing of bugs. In this work, we focus on a specific bug: Memory Leaks. We propose an automatic approach to suggest memory leak fixes in C/C++ programs saving valuable developer time.

AddressWatcher is the first attempt to use Address Sanitizer and LeakSanitizer together to suggest fixes for memory leaks. Our dynamic analysis approach was evaluated on binutils, openssh, tmux. It requires test suite to be run several times over different program paths to identify potential fix location. In 10 out of 26 real world bugs AddressWatcher was able to correctly point the free location to fix the memory leak. AddressWatcher is scalable to multithreaded applications. AddressWatcher is complementary to existing static analysis tools that fix memory leaks.

Acknowledgements

I would like to thank my supervisor Prof. Mei Nagappan, and Prof. Chengnian Sun, Prof. Meng Xu for their invaluable guidance throughout the development of this work.

Table of Contents

List of Figures

List of Tables

1 Introduction

A lot of research effort has been put into general automatic bug fixing (GABF) [**14**] [**15**] [**16**] [**17**]. In general this is done by modelling bugs as a violation of some correctness condition and using machine learning to learn from such violations. Such techniques suffer when the programs are large and the correctness conditions are not sufficient for detecting all bugs in the program. Often these techniques see limited deployment in industry and focus on trivial repairs[**22**].

The other alternative is focusing on fixing specific bugs rather than general bugs. Specific approaches are used to handle each type of bug. These approaches can be then bundled together with high effectiveness.

Memory leaks have garnered a lot of attention in the research community from detection [**1**] [**4**] [**5**] [**11**] to automatic fixing [**18**] [**19**] in recent work. Memory leaks occur when a chunk of memory is allocated in the heap with a malloc/calloc but is subsequently not freed. The languages like C/C++ do not have a custom garbage collector, so if these allocated regions are not freed they remain in application memory becoming dead weight over time. Memory leaks also lead to several critical vulnerabilities [**23**] [**24**] in software further underscoring the need to fix these as soon as they are detected.

A lot of effort has gone into detecting memory leaks, which can be broadly classified as static and dynamic approaches. Static approaches generally tend to have high false positives which can lead the developer to ignore some potentially important leaks that need to be fixed. Static approaches include tools like Saber [**1**] and Sparrow [**4**].

On the other side of the spectrum, dynamic analysis tools like LeakSanitizer (LSAN) [**5**] and Valgrind [**11**] find memory leaks with a low false positive rate but have true negatives. In other words each bug reported is valid, but they can potentially miss existing bugs because of low test suite coverage.

Other memory errors include heap and stack buffer overflows. A widely used dynamic analysis tool in industry is AddressSanitizer (ASAN) [**2**] which uses shadow memory to encode regions of valid memory and detect buffer overflows. If a read/write happens to invalid memory, ASAN detects this through compile time instrumentation, forcing a crash of the application. We intend to use this instrumentation and shadow memory for our work.

Many existing leak detection approaches only report the stack where the allocation occurs. To fix the leak, we must add a deallocation statement: a free statement, but the location of the free is not close to the allocation point. Realistically, these pointers to allocated memory are passed between functions and it often becomes difficult for the developer to find the exact point where the memory is no longer in use. If the developer inserts a wrong free statement and the memory is used thereafter, it leads to a heap use-after-free which can lead to several exploits by attackers. Hence it is important to track the use of allocated memory (read/write) along different program paths for suggesting a fix.

Other tools have experimented using both static and dynamic approaches to fix memory leaks including Leakfix [**18**]and Autofix [**19**]. Leakfix employs in-

```
#1 int main(int argc, char** argv){
#2 char *p = malloc(10);
#3 if (argc == 1) return 1; //memory leak
#4 if (argc == 2) return 1; //memory leak
#5 //use p
#6 return 0;                //memory leak
#7 }
```

Figure 1: A memory leak that requires multiple fixes.
Deallocation statements to be inserted at lines 3,4,6

terprocedural static analysis techniques while Autofix performs value flow slices on C programs to fix memory leaks. These techniques use the best of both these approaches with higher bug fixing accuracy at the cost of performance.

In this work we report our attempt to develop an approach for *suggesting* memory leak fixes specifically. We show first the complexity of suggesting a solution for these kinds of bugs. In **Figure 1** the memory allocated on Line 2 is leaked along multiple program paths through lines 3,4 and 6. Lines 3 and 4 are error paths which do not free memory before return. Line 6 similarly uses the allocated memory on non-error path but does not free before return. We must insert deallocation statements for each program path taken by the leaked object. Additionally we should not deallocate the same object twice because it will lead to a double free which can lead to undefined behaviour. The dynamic approach in this work considers how to overcome complexities involving different program paths for the same leaked object in our tool called AddressWatcher.

AddressWatcher leverages the existing infrastructure of ASAN and LSAN, with minimal changes to become an effective memory leak Bug Fixer. Our main contributions in this project include:

1. A novel approach integrating LSAN and ASAN together in AddressWatcher with minimal changes

2. Using shadow memory to tag and track Memory regions that have been leaked in previous binary runs.

3. Suggesting memory leak fixes for real world bugs in openssh, tmux and binutils

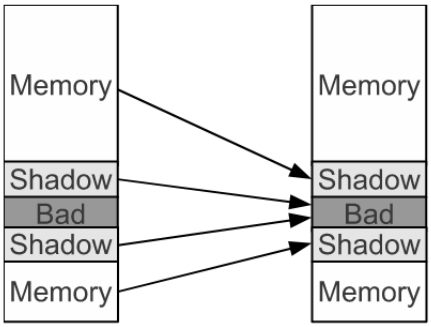

Figure 2: Address Sanitizer Shadow Memory Mapping

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
```

Figure 3: Binary Instrumention before Reads and Writes from Memory

2 Background

We will be modifying the tools ASAN [**2**] and LSAN [**5**] in this paper. A short explanation on these tools and how they work is provided for the reader. They are available in both the GCC [**20**] and LLVM toolchain [**21**].

ASAN: ASAN detects heap and stack buffer overflows during program execution. It is a dynamic analysis tool that has two phases: compile time instrumentation and dynamic runs. In dynamic runs, ASAN uses a shadow memory to encode regions of application memory which can be legally accessed. The shadow memory is typically one eighth of the application memory, mapping the state of 8 bytes of application memory in one byte of shadow memory. This shadow memory is allocated at a special offset within the program as shown in the **Figure 2.**

ASAN pads local and global variables with red zone buffers which should not be accessed by the application. During compile time instrumentation the reads and writes to memory are instrumented to incorporate additional checks. The check verifies that the application memory being read/written to is actually valid, by crosschecking the shadow memory's encoding. Such a binary instrumentation check from the ASAN paper [**2**] is shown in **Figure 3**

The right shift by 3 essentially implies divide by 8. The offset refers to the start region of shadow memory. If the shadow value is 0 then all the corresponding 8 bytes of applicaton memory are accessible and there is no memory error. However if there is a non zero value and the `AccessSize` for a given read/write does not match the legal application bytes in shadow memory then a memory

Figure 4: Redzones guarding heap allocations. These redzones contain allocation stack traces and allocator thread ID

access overflow has occurred and `ReportAndCrash` is triggered. Otherwise the read/write is allowed to occur.

It is worth noting that pointer arithmetic can lead to a wild pointer. But this will not lead to `ReportAndCrash` unless a read/write happens to that wild pointer. This is because only reads and writes are precluded by ASAN instrumentation.

LSAN: This is a sanitizer tool for memory leak detection whose leak detection phase begins after program has finished executing. It ensures that when allocation happens during a program, it is padded with extra memory called redzones, allocated for storing metadata. This metadata includes the allocation thread ID and the allocation stack itself as shown in **Figure 4**. These redzones have shadow values that are poisoned preventing them from being overwritten. When these objects are freed they are overwritten with a magic value. LSAN detects memory leaks by iterating through the heap to find out such objects that have not been freed, at the very end of program execution providing their allocation trace stored in the padded redzones.

3 Motivation and Intuition

The motivation of this work is to largely use the existing infrastructure of ASAN and LSAN with minimal modifications. The shadow memory in ASAN encodes both legal application memory regions for read/write operations and non legal regions- which are referred to as poisoned memory regions. We want to augment the existing encoded shadow memory information to also specify if the corresponding application memory is tagged or not. We tag given bytes of memory if it can be leaked.

The high level pseudocode of AddressWatcher is shown in **Algorithm 1**. AddressWatcher is interested in generating plausible leak fixes over multiple runs of the same binary, with different inputs to trigger different program execution paths through a test suite (line 10). The tests in the test suite are manually run by the developer. Since AddressWatcher collects leak information across different binary runs, we must create an external leaks database that will always remain consistent for a given binary. When a recompilation occurs the leaks database must be flushed as the database is no longer useful. To keep track of this we use an external file which stores the binary compile time which is checked against at program startup (lines 11-14). ASAN will then read from the leaks database (line 15). While running the program, the ASAN allocator checks if a new allocation matches a leaked allocation from the leaks database (lines 17-18). If it does then shadow memory of leaked object is tagged (line 19). Once shadow memory of object has been tagged, we record the points at which reads and writes happen to it through existing ASAN instrumentation (lines 22-24). At the end of each run, LSAN would store new leaks in this leaks database in the form of allocation stack traces (lines 26-30). Then we can perform stack trace comparison to identify the last point the tagged memory is used over all recorded read/write stacks per leaked object. The last read/write stack trace of these leaked objects will be updated in the leak database (line 31). We will henceforth refer to this last read/write stack for each leaked object as a LastUse stack. AddressWatcher prints this cumulative statistic over several test runs as the fix location in an output file (line 32).

A simple example is shown end to end with the philosophy of AddressWatcher in **Figure 5**. Here a buffer of size 10 is allocated to pointer p (line 1). The first character is assigned to 'a' (line 2) and second character to 'b' (line 3). When the ASAN compiled binary is run the first time, LSAN detects a leak of object by pointer p and it's allocation stacktrace is stored in the leak database. On the second binary run at line 1, AddressWatcher identifies that the allocation stacktrace matches the leak database. Hence the shadow memory of the region p points to, is tagged. Hence line 2 and 3 are tracked as use points (write stack trace) for this allocated object. At end of second binary run, AddressWatcher performs stack trace analysis and outputs the LastUse stack as line 3.

Algorithm 1 High level overview of AddressWatcher from binary standpoint

```
1: Bin: an ASAN instrumented binary with ASAN allocator (allocator)
2: Testsuite: a test suite that covers several program paths in Bin
3: Object: A region of memory surrounded by ASAN redzones
4: Objectallocation: Allocation stack of Object stored in reddzone
5: shadowmem(Object): Shadow memory of Object
6: LeakDB: External map from memory leak allocations to use points
7: compiletime: Binary compile time stored externally
8: getCurrentStack(): outputs current program stack

9: procedure AddressWatcher(Bin, Testsuite)
10:     for each test in Testsuite run by developer do
11:         if Bin compile time > compiletime then
12:             LeakDB ← φ  // Recompilation hence DB flushed
13:             Store Bin compile time in compiletime
14:         end if

15:         Load LeakDB into program memory

16:         while Running test on bin do
17:             if allocator triggered on Object then
18:                 if getCurrentStack() in LeakDB then
19:                     Tag shadowmem(Object)
20:                 end if
21:             end if

22:             if ASAN Instrumentation triggers on shadowmem(Object) then
23:                 LeakDB ← (Objectallocation, getCurrentStack())
24:             end if
25:         end while

26:         if LSAN detects new memory leaks then
27:             for each new leaked Object do
28:                 LeakDB ← (Objectallocation, φ)
29:             end for
30:         end if

31:         Analyze recorded use points per leaked object and store fix in LeakDB
32:         print detected leaks and potential fixes from LeakDB
33:     end for
34: end procedure
```

4 Design Details

In this section, we introduce implementation details of AddressWatcher, a dynamic approach to suggesting memory leak fixes. The workflow of AddressWatcher can be split into the following core components:

- **Instrumented Binary:** ASAN instruments binary during compile time
- **The Leaks Database:** A database that is consistent across varying binary runs with different input.
- **LSAN:** Storing new Memory Leaks to be tracked in the future
- **ASAN:** Reading from leaks database, and memory leak tagging and tracking

The general workflow is shown in Figure **6**. Each component is colored uniquely. We expand on each core component in the workflow below.

4.1 Binary Instrumentation

Our prototype AddressWatcher is built by modifying the GCC compiler in ASAN and LSAN. Specifically the library **libasan.so** of gcc is modified to include extra routines required for AddressWatcher. The programmer source code is compiled with the ***-fsanitize=address -g*** options with our modified gcc. This allows gcc to compile the source code and insert ASAN instrumentation before read/writes to memory as shown in **Figure 3**. We do not modify the instrumentation that ASAN currently performs. We in fact require this instrumentation and will also use it to track leaked memory. When we refer to binary henceforth we are referring to the ASAN instrumented binary.

4.2 Leaks Database

This is a database that will hold fixes to memory leaks and will update these fixes every time a new program path is found for each leaked object. Hence, this is a database that should be consistent and robust across different binary runs with different inputs. This can be achieved by the following two steps:

1. Disable Address Space Layout Randomization (ASLR): This will disable randomisation of memory regions offset.
2. Use a special directory for AddressWatcher's Leaks Database that will be guaranteed to not be used by any other program or user.

In the first point, by disabling ASLR we ensure that a deterministic program (without any randomized behaviour) will truly behave deterministic even in the memory addresses it uses. The memory regions used in stack and heap will all be the same each time it is run. Therefore a code point abstracted as a stack

```
     Program start/ASAN initialized
  #1 char *p = malloc(10);
  #2 *p = 'a';
  #3 *(p+1) = 'b';
  #4 return;
     Program End
 ->  LSAN leak detection starts      --> Store #1 as
                                         leak in database
```

(a) First binary run: LSAN stores leak in database

```
->   Program start/ASAN initialized  --> Read leak at #1
  #1 char *p = malloc(10);
  #2 *p = 'a';
  #3 *(p+1) = 'b';
  #4 return;
     Program End
     LSAN leak detection starts
```

(b) Second Binary run: Read Leaks Database at ASAN initialization

```
     Program start/ASAN initialized
-> #1  char *p = malloc(10);         --> Tag object in shadow memory
  #2 *p = 'a';
  #3 *(p+1) = 'b';
  #4 return;
     Program End
     LSAN leak detection starts
```

(c) Second Binary run: Tag memory in shadow memory for leaked object

```
     Program start/ASAN initialized
  #1 char *p = malloc(10);
->#2 *p = 'a';                      --> Record current stack #2
                                        as use of tagged memory
  #3 *(p+1) = 'b';
  #4 return;
     Program End
     LSAN leak detection starts
```

(d) Second Binary run: Track object in shadow memory

```
     Program start/ASAN initialized
  #1  char *p = malloc(10);
  #2  *p = 'a';
->#3  *(p+1) = 'b';                --> Record current stack
                                       #3 as use of tagged memory
  #4  return;
     Program End
     LSAN leak detection starts
```

(e) Second Binary run: Track object in shadow memory

```
     Program start/ASAN initialized
  #1  char *p = malloc(10);
  #2  *p = 'a';
  #3  *(p+1) = 'b';
  #4  return;
     Program End
->   LSAN leak detection starts     --> #3 > #2.
                                        So store #3 as solution for leak at #1
```

(f) Second Binary run: Store solution in leak database

Figure 5: Sample example for AddressWatcher on two binary runs of same program

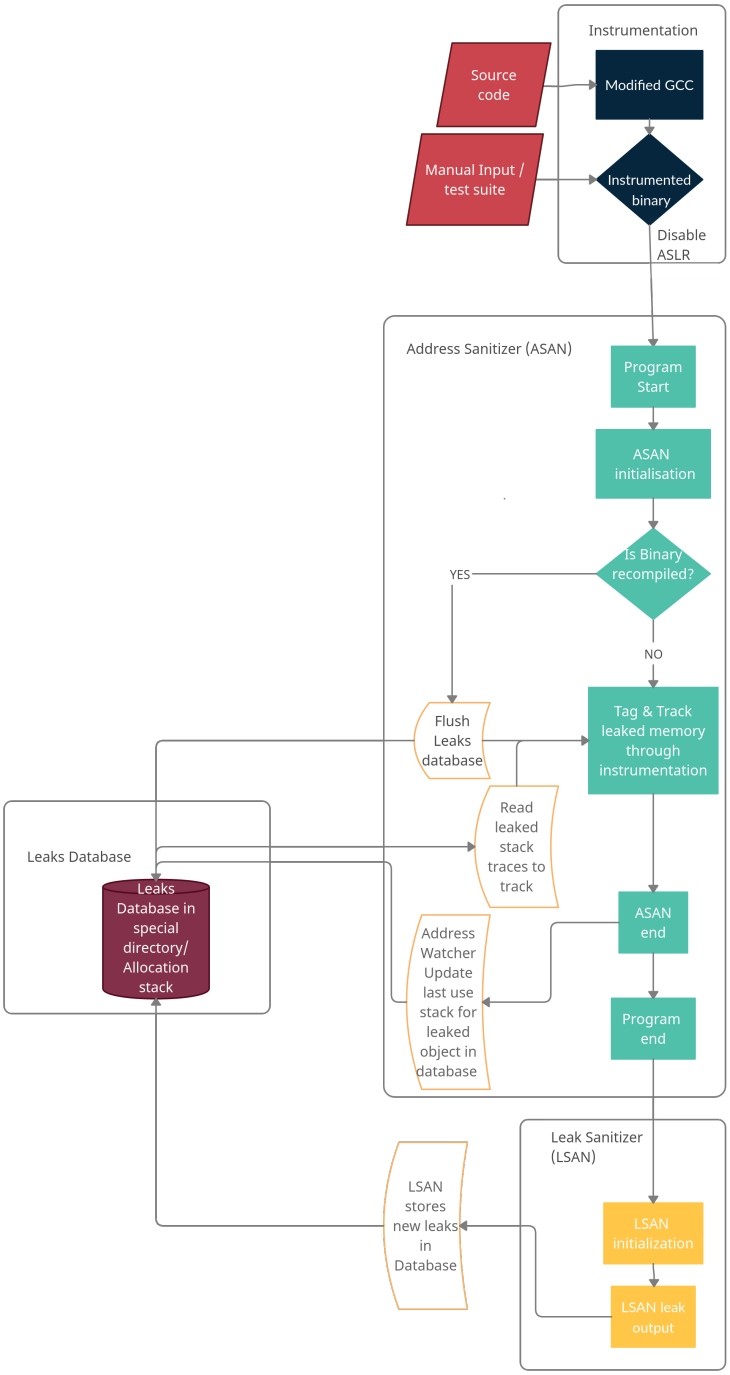

Figure 6: Address Watcher Workflow. Each core component is labelled in a different colour. LSAN related tasks are yellow. ASAN related tasks are green. User Inputs are red. The Leaks Database is purple. Instrumented binary is blue.

should be valid over several binary runs. This makes stack traces consistent over any binary run.

Additionally in the second point we are ensuring that a special directory is available to Address Watcher which cannot be used by any user/program. This is essential because any corruption in the Leaks Database can render subsequent tagging and tracking impossible.

ASAN currently generates its logs for memory errors in a path set by the user in an environment variable ASAN_OPTIONS under log_path parameter. In the same directory set by user, AddressWatcher can create a hidden directory **.awlogs** which cannot be modified directly by other users and programs. We can then store the Leaks Database in this special hidden directory. This will realize the second point for enforcing robustness of the leak database.

It is important to decide the way data is stored in the Leaks Database. It is worth noting that in the current prototype the entire database is read into memory during program initialization. A call is made from instrumented binary at startup to `asan_init` in ASAN library **libasan.so**. Here along with ASAN initialization the entire leaks database is read using C++ **mmap()** routine. The current database is simply a text file with serial ordering of allocated stacks and LastUse stacks for leaked objects. This means that we do not essentially query this database with different constraints. After we have updated the relevant read/write traces per leaked object in memory we currently overwrite the leak database with the leaks and fixes in no particular order.

It is also important to realize that we would have a leaks database in this hidden directory for every distinct binary being tested with AddressWatcher. Every binary is uniquely determined by it's name and directory path. Threads of the same process belong to the same binary and hence update the same leaks database. Multithreaded processes of same binary must have a lock on writing to the Leaks Database to ensure its consistency. This infrastructure is already present within ASAN and can be leveraged easily. AddressWatcher is thread safe because the underlying ASAN infrastructure is thread safe and interactions with the leak database are performed using locks.

The data within the leaks database is valid only so long as the binary in question has not been recompiled. On recompilation the leaks can be fixed or other major code realignments could have taken place. This renders the data in the Leaks Database void. For this purpose we also label each Leaks Database with the compile time of it's binary. So when ASAN library **libasan.so** reads this data it can crosscheck its compile time with the time associated with the Leaks Database. If there is recompilation the Leaks Database must be flushed and AddressWatcher will have to accrue data from the beginning (relabeling the new empty database with the new binary compile time). This is an important aspect of automating AddressWatcher and ensuring leaks database integrity on recompilation.

4.3 LeakSanitizer (LSAN)

This section describes the function of LSAN in AddressWatcher. LSAN performs a leak detection phase at the end of program execution. When objects are allocated in heap through allocator, extra size is allocated for metadata redzone which includes allocation stack trace and allocator thread Id. When an object is freed it is overwritten with a magic value. Hence LSAN can iterate over the heap at program end and identify leaked objects and their allocation stack traces. These are output in the log_path directory for the user to view. We modify the code of LSAN to additionally write these allocation stacks of these leaked objects to the Leaks Database in the **.awlogs** directory.

LSAN essentially records new leaks in Leaks Database. AddressWatcher depends on this to effectively utilize subsequent tests on the binary in the future for leaks tracking.

A naive way to approach the problem is to treat all allocated objects as leaked objects and track these for solutions/fixes. AddressWatcher rather seeks to amortize the cost of finding a solution/fix across several tests in a test suite. Only when LSAN detects a memory leak does AddressWatcher track that allocated memory region in *subsequent* tests/runs. This boosts performance compared to the naive solution.

4.4 AddressSanitizer (ASAN)

ASAN is the most important core component of AddressWatcher. It is responsible for:

1. Tagging leaked memory in shadow values
2. Tracking read and write traces of leaked objects

4.4.1 Tagging Leaked Memory

At program initialization, ASAN is initialized through a binary call to `asan_init` in ASAN library **libasan.so**. Here the shadow memory is set up along with a host of routines that read the `ASAN_OPTIONS` environment variable to configure ASAN for that run. At this point AddressWatcher also reads from the Leaks Database all the leaked allocation stacktraces in all the previous runs. This list of stacks is read into memory. This list of stacks is then sorted with respect to stack depth and the values within the stack. This sorted list of allocation stacks makes it easy for a O(logn) check if an allocation stack trace exists.

The ASAN allocator is modified in AddressWatcher. Whenever an allocation happens, we compare the stack trace of the application at the given point to the stack trace in our sorted list. If the stack trace does not exist then we continue just like vanilla ASAN would.

If there is a match in our sorted list, it implies that there has been a previous run over the same binary where this program point was passed and this object was leaked. AddressWatcher hence calculates the shadow addresses for the

```
UpdateTaggedObject(AllocationStack)
{
    AVL Tree T
    CurrentStack = UnwindProgramStackTrace()
    T.InsertorUpdate(AllocationStack,CurrentStack)
}


ReportAndCrash(Addr,MemAccessSize)
{
    if (MemCorrupted(Addr,MemAccessSize))
        CrashAndPrintReport() //Buffer Overflow -Crash

    shadowAddr = MemToShadow(Addr,MemAccessSize)

    if (*shadowaddr & 0xf0 == 0xe0)
    {
        // Tagged Memory
        ObjectAllocationStack = MemToObject(Addr)
        UpdateTaggedObject(ObjectAllocationStack)
    }
    return // Continue execution
}
```

Figure 7: Simplified ReportAndCrash() pseudocode: UnwindProgramTrace retrieves current program stack at execution. MemToObject retrieves the object corresponding to tagged memory. MemCorrupted verifies if memory and access size cause a buffer overflow

object in this case. The higher order byte of shadow values are set to **0xe** in our prototype. This essentially tags this memory region for subsequent tracking. The shadow encoding for different types of application memory is shown in **Table 1**.

4.4.2 Tracking Leaked Memory

There are two functions critical for tracking tagged objects. The pseudocode for these routines is in **Figure 7**. They are explained in depth below:

Invoking ReportAndCrash(): AddressWatcher hinges on the idea of using `ReportAndCrash()` in **Figure 3** for effective tracking. This function is invoked only when code passes through the shown binary instrumentation before a memory read/write. Leaked objects would have their higher order bits set in the value of `ShadowAddr` in **Figure 3** from the previous subsection. Since

`k` is negative as a signed byte, both the sub expressions in the if condition evaluate to false. **Hence ReportAndCrash will be called for every tagged memory object**. This implies that `ReportAndCrash()` in **libasan.so** will be called in either of the two scenarios:

- A memory access occurs (Heap/Stack overflow) just like vanilla ASAN OR
- Tagged memory region is read/written

This routine `ReportAndCrash()` is modified to check the higher order byte to differentiate the above two cases. If it is a buffer overflow we crash just like ASAN normally would.

If it is a tagged memory region, AddressWatcher does not crash. Once we have identified a read/write to middle of tagged memory chunk, we must iterate backwards to the beginning of the object, where the metadata redzone exists as in **Figure 4**. This metadata includes allocation stack trace and allocator thread ID which can accurately identify the object the tagged memory corresponds to. Once the object is identified we call `UpdateTaggedObject()` with the object allocation stack as argument. We then return to application code from library code, to *continue* execution.

Invoking UpdateTaggedObject(): For efficient stack tracing an AVL Red Black Tree is created that can be used only by this routine. Every node in the AVL tree has two data fields:

1. The allocation stack trace of leaked object it refers to.
2. List of read/write stack traces of this object in current execution.

This AVL Red black tree is initialized with no objects at ASAN initialization (when the leaked objects allocation Stacktraces are read). The primary purpose of this query optimized tree is fast update of an existing node and fast insertion of a new node, both in O(logn) time. This optimization is possible because the height of the tree is always balanced. Without a fast data structure it will be expensive to track several leaked objects within a given program execution.

If this routine finds a completely new tagged object read/written to, we insert a new node in the tree. We then initialize the head of the second field-the list, with the current program stack. If the tagged object exists in the tree, we append the current program stack to the list in the second data field. The second data field will be updated as we see more reads and writes for a given leaked object.

```
                0x00100
0x00201         0x00200
0x00300   >s    0x00300
0x00400         0x00400
Stack 1         Stack 2
```

Figure 8: Example of comparison operator >s for comparing stacks. Here from bottom up: 0x00400 is equal, so we move one level up. Similarly 0x00300 is equal. But in the next level 0x00201 > 0x00200. Hence the first stack is greater than second stack by this operator >s

5 Identifying Fix Location

We introduce a new comparison operator **>s** for the purpose of finding the last point at which reads and writes have happened over several runs. We define A **>s** B by comparing stack values in a bottom up fashion. When two stack values are equal we then compare the next higher level of the stacks. A sample example of the **>s** operator is shown in **Figure 8**. The intuition behind such a comparison operator is that it would identify the code point which occurs the last in execution, without any loops and goto statements. **Figure 9** shows a loop example where **>s** produces the wrong LastUse point. Source code without loops/while/goto will always have execution proceed towards memory addresses that are larger in value. Hence by comparing stacks in a bottom up fashion we can arrive at an approximate point in the code which is the last read/write over all executions.

At the end of the program we must iterate through all leaked objects in the AVL tree modified by the routine `UpdateTaggedObject()`. For each leaked object we must sort through the read/write stacktraces and merge them towards a point where the developer can insert a free statement using the **>s** operator. This fix location in our tool is a cumulative statistic that improves as more tests are introduced and more program paths are covered. We also compare with the fix from previous binary runs for each object, from the Leaks database. This resultant stack is the LastUse stack over all binary runs for that leaked object. This is stored to the Leaks Database as the fix for the given memory leak.

We only suggest one code point where a deallocation statement must be added. However there can be cases as in **Figure 1** where multiple fixes are required. In such cases we aim to provide one correct fix.

It is worth noting that the fix that AddressWatcher suggests can potentially lead to use-after-free if the test suite does not have high code coverage. If the code coverage is sufficiently large then a use-after-free cannot happen because we output the last read/write trace of the leaked object. Regardless of the test suite coverage, we provide the developers with a tool that can significantly speed bug fixing with precise information on read/write traces of the leaked object.

```
#1 char *p = malloc(10);
#2 int i = 1;
#3 while(i++ < 10){
#4    if (i == 8){
#5        //use p
#6        exit(0); //memory leak
#7    }
#8    //use p
#9 }
```

Figure 9: Example of comparison operator >s producing wrong Last Read-/Write code point in program as a solution. A program will run through line 8 several times before exiting abruptly at line 6 after a use at line 5. In such cases >s will produce line 8 as correct fix, which is wrong. Stronger static analysis is required to produce line 5 as solution

Table 1: Shadow encoding in ASAN for 8 to 1 shadow mapping

Application bytes	**ASAN encoding**	**AddressWatcher encoding**
1 byte addressable	0x01	0xe1
2 byte addressable	0x02	0xe2
3 byte addressable	0x03	0xe3
4 byte addressable	0x04	0xe4
5 byte addressable	0x05	0xe5
6 byte addressable	0x06	0xe6
7 byte addressable	0x07	0xe7
8 byte addressable	0x00	0xe0

Table 2: Total bugs in each repository

Repository Name	**Number of memory leak bugs**
binutils	8
tmux	9
openssh-portable	10

Table 3: Benchmark for AddressWatcher. Legend for Bug Classification: Error Path - fix along error conditions which do not read leaked objects before returning, Loop Path - fix along loop , Code organization - seperate functions used for deallocations, Weak test suite- low test suite coverage covering all leaked program paths, Compiler Optimization - Compiler optimizes away certain read-/writes important to detecting fix

Repository	**Buggy Github Parent ID**	**Github memory leak fix commit ID**	**AW Failure reason**	**AW Fix (Y/N)**	**Memfix Fix (Y/N)**
binutils	a506516	be74fad95edc8827516e144cf38d135b503249cd	-	Y	N
	2c244f9	3cfd3dd0956fe854a07795de12c1302ecabbd819	-	Y	Y
	52a93b9	a26a013f22a19e2c16729e64f40ef8a7dfcc086e	Loop path	N	N
	e13cb30	7ed1acafa0b5d135342f9dcc0eb0387dff95005a	Code Organization	N	N
	2f5404b	f978cb06dbfbd93dbd52bd39d992f8644b0c639e	Loop Path	N	N
	ad36c6c	3f2a3564b1c3872e4a380f2484d40ce2495a4835	-	Y	N
	c42608e	848ac659685fba46ce8816400db705f60c8040f7	-	Y	N
	9d2cdc8	aba19b625f34fb3d61263fe8044cf0c6d8804570	Error Path	N	Y
tmux	7ba5ad4	c363c236aaea5b7a879493d8f3c85bead546f063	-	Y	Y
	ae1a6c2	1e0eb914d945e0f287716d56669d0de409e86e59	-	Y	Y
	c8ecbf3	2c9bdd9e326723fb392aed4d8df12cba7ef34f1f	Error Path	N	Y
	54bcaab	d566c780e54010112d499707cd80a594144d1a89	-	Y	Y
	40fefe2	933929cd622478bb43afe5906706 13da2e9ff359	Error Path	N	Y
	695a591	7340d5adfdc8cc6d845a373f3e0d59bfd10a45d1	Error Path	N	N
	540f0b3	189017c078b7870c18ced485c1fd99f65fcc4801	Loop Path	N	N
	871b83c	5acee1c04ed38afd6a32da4a66e6855ccdc52af3	Error Path	N	N
	69b7c49	6daf06b1ad61f67e9f7780d787451b9b5f82dd43	Error Path	N	N
openssh-portable	6d5a41b	b2afdaf1b52231aa23d2153f4a8c5a60a694dda	Compiler Optimization	N	N
	7d6c036	66d2e229baa9fe57b86	Error Path	N	N
	e6b9503	a63cfa26864b93ab6afefad0b630e5358ed8edfa	Error Path	N	Y
	7ad8b28	4f7cc2f8cc861a21e6dbd7f6c25652afb38b9b96	Weak test suite	N	Y
	f948737	64a89ec07660abba4d0da7c0095b7371c98bab62	Error Path	N	N
	b1ba15f	165bc8786299e261706ed60342985f9de93a7461	Error Path	N	N
	a5103f4	aae07e2e2000dd318418fd7fd4597760904cae32	-	Y	Y
	0cca17f	e52a260f16888ca75390f97de4606943e61785e8	Weak test suite	N	Y
	534b2cc	393920745fd328d3fe07f739a3cf7e1e6db45b60	-	Y	N
	467b00c	0d6771b4648889ae5bc4235f9e3fc6cd82b710bd	-	Y	Y

6 Implementation: Compatibility with ASAN and LSAN

To create AddressWatcher we required to modify certain sections of existing GCC code in the sanitizer directory used while compiling **libasan.so** library:

- `CrashAndReport` function
- Add functionality for initializing AddressWatcher. While initializing we check for binary recompilation to enforce integrity of leaks database. This is added at ASAN initialization
- Modify ASAN allocator to tag memory which is identified as a leaked object

In total 27 files were changed with 2,691 additions and 1,895 line deletions requiring minimal modifications. We mention the shadow encoding used by AddressSanitizer for shadow memory in **Table 1**. The only changes we require are for the addressable bytes. The modifications in shadow value representations do not change the behaviour of ASAN in any way, because they are appropriately handled in `ReportAndCrash`.

7 Performance and fix accuracy

Our prototype is constructed so as to optimize performance on every test suite run. To output plausible fix results the test suite must be run atleast twice to obtain the best results. This is because one run is required to identify the leak itself (through LSAN at end of execution), and the second run is required to track it's unique code path. We can also consider limiting the number of leaked objects that AddressWatcher is willing to track for a given binary.

We evaluate AddressWatcher on fixes suggested compared with the developer's fix for real world memory leaks. When we evaluate the fixes we assume that the testsuite has been run twice to provide the most accurate fix that AddressWatcher can possibly suggest.

We compare our fix results with Memfix **[13]** which is an open source tool for fixing Memory Leaks through static analysis. Memfix fixes memory leaks by identifying leaked objects statically and all the program paths that these leaked objects need to be alive in. Then the problem of a fix is reduced to a Exact Cover problem essentially stating that minimum frees must be inserted to cover all leaked paths. This problem is solved by a SAT solver **[10]**. Then all frees in these paths already present in source are removed, and the new free statements are added. This potentially fixes double-free and heap-use-after free along with memory leaks itself.

AddressWatcher was tested on the curated benchmark provided by Memfix especially for testing memory leak fixers. The benchmarks were created using CIL (C Intermediate Language) **[25]** which compiles complex source code with real world memory leak bugs into few core constructs with clean semantics. It includes memory leak bugs in the open source repositories: openssh, binutils, tmux. We first check these bugs in the github parent and ensure that the developers fixing these issues labels them as 'Memory Leaks'. The ground truth for these bugs are the locations where the programmer inserts a free statement. We then compare this ground truth with the solutions suggested by AddressWatcher and Memfix.

The results are shown in **Figure 10** over 27 bugs. The distribution of bugs across repositories are shown in **Table 2**. Detailed bug information is available in **Table 3**.

To better understand the intersection of bugs between AddressWatcher and Memfix, we show the results in **Figure 11**. Each bug is counted in the figure only once. This means that AddressWatcher has fixed 5 bugs independantly that Memfix could not solve. But AddressWatcher also solves 5 bugs that Memfix can also solve. Hence AddressWatcher has fixed a total of 10 bugs with an accuracy of 38%. 10 bugs in the repositories were not fixed by both AddressWatcher and Memfix.

AddressWatcher fails to provide developer's solutions on program error paths. We provide an example in **Figure 12**. Here an error path is triggered because the user does not supply the correct arguments to the program. However there is no read/write to the leaked object in the error handling routine and hence we cannot suggest the developer's fix in this case.

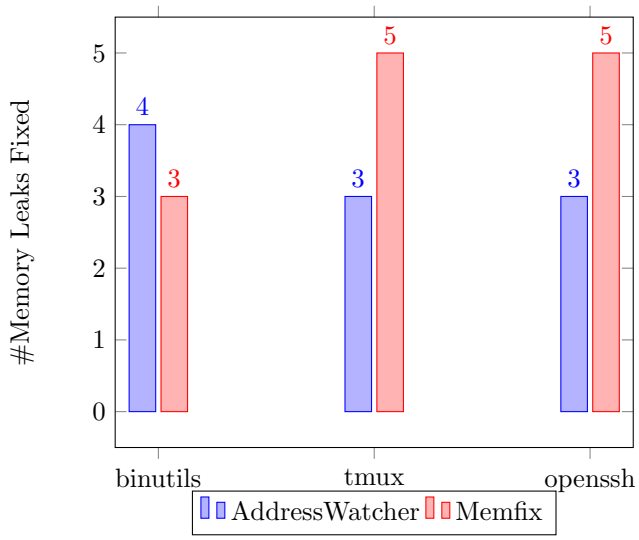

Figure 10: Distribution of bugs correctly fixed over different repositories

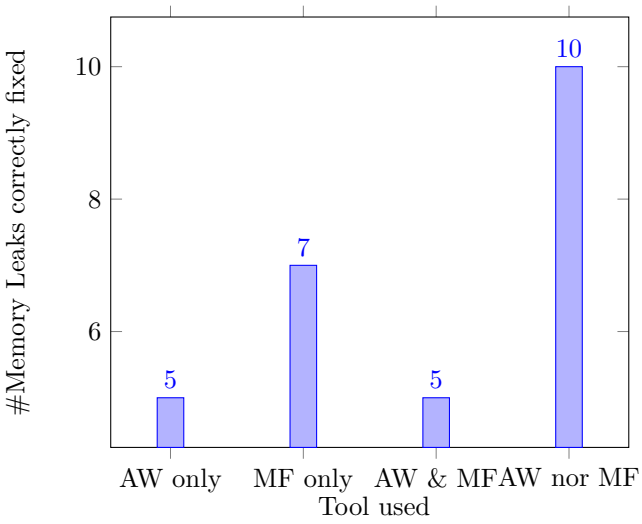

Figure 11: Distribution of bugs between Memfix (MF) and AddressWatcher(AW)

```
#1 char *p = malloc(10);
#2 if( argv == NULL) goto ERROR; //memory leak
#3     // use p
#4 return 0;
#5 ERROR:
#6 //error handling routine
#7 exit(1);
```

Figure 12: Example of case where AddressWatcher fails to fix memory leak. The free statement must be inserted at #6 before the abrupt exit. However even if a test suite triggers these error paths, the pointer p is not read or written in error checking. Hence AddressWatcher cannot suggest a fix along these error paths

All of the 17 cases AddressWatcher fails on can be split in the below 4 categories:

- Error paths in the program code or abrupt returns where leaked objects are not accessed - **10 cases**
- Loops in which leaked objects are accessed. Hence >**s** operator can potentially provide the wrong fix to a memory leak without static analysis - **3 cases**
- Code organization: A seperate routine is used by the developer that is used to free all allocated objects irrespective of where it is last used - **1 case**
- Weak test suite with low coverage which does not exercise all code paths for leaked object -**2 cases**
- Certain read/ writes are not instrumented by ASAN because they are statically determined to not cause heap overflows or the read/write is removed by register optimizations. In these cases AddressWatcher cannot track the read/write stack trace - **1 case**

Another interesting point to consider is that Memfix alone fixes 7 bugs independantly of AddressWatcher. These fixes largely contribute to several error path fixes that AddressWatcher cannot find. On the other hand AddressWatcher finds 5 fixes independently of Memfix, because Memfix suffers from path explosion in these particular cases(timeout). These tools are hence complementary in nature and can be used together to fix 17 real world Memory Leaks together.

We finally note that the proposed fix from AddressWatcher is just one plausible fix which may or may not be finally accepted by the developer. For example, the developer may choose not to adopt the fix at the last use point from a code design perspective, preferring to have all deallocations in one routine at the end.

8 Limitations And Future Work

AddressWatcher currently has several limitations which must be addressed in future work.

Currently, AddressWatcher does not have the capability to detect data corruption in leaks database. Checksum information or parity bits can be stored alongwith the Leaks Database to detect corruption. When corruption is confirmed the entire Leaks Database can be flushed.

AddressWatcher is incapable of suggesting leak fixes on error paths that do not use leaked variables or error paths not covered by test suite. For such cases we can also use light weight static analysis to find out the true last read/write access. This would also replace the **>s** operator by matching the collected read/write traces with the source code and deriving the correct fix. This will provide correct fixes even in the case of goto/loop statements.

AddressWatcher currently only suggests one fix for a memory leak. However there can be cases where a single leaked object requires insertion of multiple frees as in **Figure 1**. We must handle such cases through static analysis on read/write traces of leaked objects (mapping to source code). AddressWatcher can also be combined with Memfix to suggest leak fixes on program error paths in future.

AddressWatcher is also fundamentally limited by the code coverage of the test suite. If the test suite does not cover all the paths of a leaked object then a plausible solution cannot be suggested. In future automatically generated test suites that execute all paths using leaked objects can be considered. AddressWatcher cannot function reliably by covering all leaked program paths in randomized programs.

Another important limitation of AddressWatcher is compiler optimizations. In certain cases if the program is trivial or the result of a certain chain of memory access can be statically predicted they can be removed by the compiler, or transformed to a smaller set of memory access with register optimizations. In this case, the binary does not exactly reflect the source code behaviour in terms of memory access and hence AddressWatcher cannot provide an accurate solution in these cases. This is because AddressSanitizer only instruments read-/writes from memory but not modifications within registers. However taking register optimizations into consideration we can say that the suggested fix point of AddressWatcher will be close to the actual fix that is required. These optimizations were seen to prevent AddressWatcher from providing the best fix in only one case tested.

9 Related Work

Static approaches for memory leak detection: Saber [**1**] employs a full sparse value flow graph that captures define and use chains and value flow via assignments for all memory locations. Their static analysis approach has detected over 83 memory leaks in SPEC 2000 C programs with a False positive detection rate of 19%. FastCheck [**4**] is another static analysis tool for memory leaks which is based on Value flow analysis.

On the other hand, Sparrow [**3**] converts each procedure's memory behaviour into a method summary that is used in analyzing callsites for leaked memory. Sparrow reports 81 bugs on the SPEC 2000 C benchmark with a 16% false positive rate.

Most static approaches are neither sound (missing bugs) nor complete (high false positives). Fastcheck and Saber bounds loops and recursion to atmost one iteration and path correlations are ignored. These approaches have imprecision in handling pointer arithmetic as well.

Dynamic approaches for memory leak detection: LeakSanitizer [**5**] is an example of a dynamic analysis approach for detecting memory leaks by replacing the allocator and using magic values for freed memory. The accuracy of such approaches are dependant on the test coverage of the test suite implying that it can miss out on several inherent bugs in source program. However these approaches have generally zero false positives.

AddressWatcher on the other hand suggests memory leak fixes. Perhaps as future work, the memory leaks detected by all these tools can be transferred in a common format to the leak database so that AddressWatcher can tag and track them.

Dynamic approaches for Memory Errors detection including heap and stack overflows: Valgrind [**11**] Memcheck keeps track of all heap blocks allocated and so can directly identify memory leaks. Valgrind also detects several other memory errors through synthetic execution and by shadowing memory and registers. But this approach suffers from significant performance overhead due to a parallel synthetic execution.

AddressSanitizer (ASAN) [**2**] detects memory corruption vulnerabilities including stack and heap buffer overflows with the help of shadow memory and compile time instrumentation. However these approaches face several limitations. Firstly it has over 3x memory overhead degrading performance and cannot detect non contiguous memory violations. Furthermore it cannot detect sub object buffer overflows. AddressWatcher focuses only on fixing memory leaks and leverages ASAN infrastructure to achieve it.

Static approaches for memory leak fixing: Memfix[**13**] is a static analysis approach to fixing memory leaks, heap use-after-free and double frees at the same time. All program paths of a leaked object are identified in the first phase. Then the bugfix is modelled as a Exact Cover problem where minimum frees must be placed to plug all the leaked paths. A solution is generated using a SAT solver. Then all generated deallocation statements are inserted along the program paths. However Memfix suffers from timeouts on program path

explosion and recursion. We compare AddressWatcher with Memfix because the artifacts are readily available and easy to use.

Dynamic approaches for memory leak fixing: LeakPoint [**12**] is the work closest in approach to AddressWatcher. It is a dynamic analysis framework that provides the location where reference to allocated object is lost or last used in a dynamic path. It hence suffers from the same limitations as AddressWatcher. However AddressWatcher is novel in that it leverages existing performance optimized infrastructure of ASAN and LSAN which is widely deployed in the industry, with minimal changes.

Garbage Collectors: The most widely used dynamic approaches to memory leak fixing is garbage collection [**6**] [**7**] [**8**] [**9**]. However this is very hard in languages like C/C++ which do not have a clear distinction between pointers and data. This can lead to hidden pointers through type casting from pointers to integers, preventing garbage collectors from knowing when an object is truly not referenced by any pointer. Only when a garbage collector knows an object is not referenced by all pointers and data can the object be freed. This in principle leads to significant performance overhead. AddressWatcher in comparison does not automatically free leaked memory at run time but only suggests to the developer possible free locations.

10 Conclusion

In this work, we presented a new dynamic analysis framework for suggesting memory leak fixes automatically. AddressWatcher can be easily merged with AddressSanitizer and LeakSanitizer with minimum modifications. We show that our novel solution solves 10 out of 26 real world bugs in a host of open source projects including openssh, binutils and tmux. AddressWatcher is complementary to existing static analysis memory leak fixers like Memfix.

11 Artifacts

The artifact is available at: `https://github.com/darkforce392/gcc`

This artifact is forked from master GCC branch. All the modifications are made into two directories: gcc/ and libsanitizer/. The following files are modified:

1. **asan_allocator.cpp:** This modfies the allocator to tag shadow memory appropriately when leaked object is identified.
2. **asan_report.cpp:** This is the file where **ReportAndCrash()** is modified
3. **lsan_common.cpp:** This is the file where leaks and related information is written to Leaks database.
4. **asan_rtl.cpp:** Initialize AddressWatcher correctly by reading from database.
5. AddressWatcher Results is a pdf file that contains the results in a tabular form. It is in the main directory of the repository
6. Other files have been added and are used exclusively by AddressWatcher. These are called from the above modified regions of code.

This modified gcc will have to be built by first building the dependencies like mpfr, mpc and gmp. Then these locations are specified as options while building gcc.

References

[1] Sui, Y., Ye, D. and Xue, J., 2012, July. Static memory leak detection using full-sparse value-flow analysis. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (pp. 254-264).

[2] Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D., 2012. Address-sanitizer: A fast address sanity checker. In 2012 USENIX Annual Technical Conference (USENIXATC 12) (pp. 309-318).

[3] Jung, Y. and Yi, K., 2008, June. Practical memory leak detector based on parameterized procedural summaries. In Proceedings of the 7th international symposium on Memory management (pp. 131-140).

[4] Cherem, S., Princehouse, L. and Rugina, R., 2007, June. Practical memory leak detection using guarded value-flow analysis. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 480-491).

[5] Leak sanitizer: https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

[6] Jones, R. and Lins, R., 1996. Garbage collection: algorithms for automatic dynamic memory management. John Wiley and Sons, Inc..

[7] Boehm, H.J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. Software: Practice and Experience, 18(9), pp.807-820.

[8] P. Wilson, "Uniprocessor garbage collection techniques,"in Memory Management. Springer Berlin Heidelberg,1992, vol. 637, pp. 1–42.

[9] H.-J. Boehm, "Bounding space usage of conservative garbage collectors," inPOPL '02, 2002, pp. 93–100

[10] "Satisfiability and SAT Solvers." SAT and SAT Solvers, cse.buffalo.edu/ erdem/cse331/support/sat-solver/index.html.

[11] Nethercote, N. and Seward, J., 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. ACM Sigplan notices, 42(6), pp.89-100.

[12] Clause, J. and Orso, A., 2010, May. Leakpoint: pinpointing the causes of memory leaks. In 2010 ACM/IEEE 32nd International Conference on Software Engineering (Vol. 1, pp. 515-524). IEEE.

[13] Lee, J., Hong, S. and Oh, H., 2018, October. Memfix: static analysis-based repair of memory deallocation errors for c. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 95-106).

[14] Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B. and Zeller, A., 2010, July. Automated fixing of programs with contracts. In Proceedings of the 19th international symposium on Software testing and analysis (pp. 61-72).

[15] Arcuri, A. and Yao, X., 2008, June. A novel co-evolutionary approach to automatic software bug fixing. In 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence) (pp. 162-168). IEEE.

[16] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S., 2009, May. Automatically finding patches using genetic programming. In 2009 IEEE 31st International Conference on Software Engineering (pp. 364-374). IEEE.

[17] Qi, Y., Mao, X., Wen, Y., Dai, Z. and Gu, B., 2012. More efficient automatic repair of large-scale programs using weak recompilation. Science China Information Sciences, 55(12), pp.2785-2799.

[18] Gao, Q., Xiong, Y., Mi, Y., Zhang, L., Yang, W., Zhou, Z., Xie, B. and Mei, H., 2015, May. Safe memory-leak fixing for c programs. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Vol. 1, pp. 459-470). IEEE.

[19] Yan, H., Sui, Y., Chen, S. and Xue, J., 2017. AutoFix: an automated approach to memory leak fixing on value-flow slices for C programs. ACM SIGAPP Applied Computing Review, 16(4), pp.38-50.

[20] GCC toolchain https://gcc.gnu.org/

[21] LLVM toolchain https://llvm.org/

[22] Kirbas, S., Windels, E., McBello, O., Kells, K., Pagano, M., Szalanski, R., Nowack, V., Winter, E.R., Counsell, S., Bowes, D. and Hall, T., 2021. On the introduction of automatic program repair in Bloomberg. IEEE Software, 38(4), pp.43-51.

[23] CWE-401 Memory leaks: https://cwe.mitre.org/data/definitions/401.html

[24] Mitre database of memory leak vulnerabilities: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=memory+leak

[25] CIL: http://people.eecs.berkeley.edu/ necula/cil/