

# AddressWatcher: Sanitizer-Based Localization of Memory Leak Fixes

Aniruddhan Murali<sup>1</sup>, Student Member, IEEE, Mahmoud Alfadel<sup>1</sup>, Member, IEEE, Meiyappan Nagappan<sup>1</sup>, Meng Xu<sup>1</sup> Member, IEEE and Chengnian Sun<sup>1</sup>, Member, IEEE

**Abstract**—Memory leak bugs are a major problem in C/C++ programs. They occur when memory objects are not deallocated. Developers need to manually deallocate these objects to prevent memory leaks. As such, several techniques have been proposed to automatically fix memory leaks. Although proposed approaches have merit in automatically fixing memory leaks, they present limitations. Static-based approaches attempt to trace the complete semantics of memory object across all paths. However, they have scalability-related challenges when the target program has a large number of leaked paths. On the other hand, dynamic approaches can spell out precise semantics of memory object only on a single execution path (not considering multiple execution paths). In this paper, we complement prior approaches by designing and implementing a novel framework named *AddressWatcher*. *AddressWatcher* allows the semantics of a memory object to be tracked on multiple execution paths as a dynamic approach. *Addresswatcher* accomplishes this by using a leak database that is designed to allow storing and comparing different execution paths of a leak over several test cases. We conduct an evaluation of *AddressWatcher* over five popular open-source packages, namely *binutils*, *openssh*, *tmux*, *openssl* and *git*. In 23 out of the 50 examined real-world memory leak bugs, *AddressWatcher* correctly points to a free location to fix memory leaks. Moreover, to demonstrate the real-world impact of *AddressWatcher*, we submitted 25 pull requests (PRs) to 12 popular open-source project repositories. These PRs targeted the resolution of memory leaks within these repositories. Among these, 21 PRs were merged, addressing 5 open GitHub issues. In fact, a critical fix prompted a new version release for the *calc* repository, a program used to find large primes. Furthermore, our contributions through these PRs sparked intense discussions and appreciation in various repositories such as *coturn*, *h2o*, and *radare2*, highlighting the significant impact of *AddressWatcher*.

**Index Terms**—Memory leak, Dynamic analysis, Vulnerability

## 1 INTRODUCTION

Memory leaks are common bugs in programming languages like C/C++. They mainly occur when dynamically allocated objects are not deallocated. Programming languages like C and C++ do not have automatic garbage collection, instead, they rely on developers to manually deallocate memory objects. Due to such a manual process, developers may forget to deallocate an object, causing a memory leak.

Memory leaks may have a large negative impact on software systems if not carefully examined and fixed. In fact, memory leaks are direct sources of security vulnerabilities. Attackers can utilize a memory leak to launch a denial of service (DoS) by crashing or hanging the program and taking advantage of other unexpected program behaviour resulting from low memory condition [1]. Recently, a number of memory leak vulnerabilities have been disclosed in Linux kernel (e.g., CVE-2022-27819 [2], CVE-2017-10810 [3]). Such vulnerabilities in the kernel had severe consequences on system stability and availability [4, 5, 6].

Fixing memory leaks manually is often time-consuming and error-prone for developers [7]. Hence, prior work focused on designing and implementing techniques to address challenges of automatically fixing memory leak bugs [7, 8, 9, 10, 11]. Many of these prominent techniques leverage static and dynamic approaches to fixing memory

leak bugs, although very few of these approaches have been open-sourced [7, 8] (both of them being static approaches).

An exemplary static analysis tool is *Memfix* [7]. *Memfix* identifies all paths involving an allocated memory. It models the problem of identifying a set of deallocation statements on these identified paths as an exact cover problem. It then uses a SAT solver to find the solution to the exact cover problem [12]. The solution suggested by *Memfix* is always a safe fix. However, *Memfix* cannot resolve program paths in the presence of function pointers and recursion, and it errors out when there is an explosion of leaked program paths.

Furthermore, prior work has proposed dynamic analysis techniques to mitigate the problem of over-approximation by static approaches. *LeakPoint* is one example of such an approach, which is a dynamic analysis tool that performs taint propagation on leaked objects [11]. It identifies last-use sites of leaked objects and suggests candidate sites for leak fixing. One limitation of *LeakPoint* is that the fix is limited to considering a given execution path.

In this paper, we complement prior approaches (e.g., static approaches like *Memfix*) by proposing a new open-source dynamic approach for memory leak fixing, called *AddressWatcher*. Our approach aims to automatically identify locations where a memory leak should be fixed. *AddressWatcher* is an iterative process that is designed to refine its fixes over several test runs as more memory leak paths are uncovered. *AddressWatcher* achieves this by cross-linking the runtime behavior of the same memory object across a fleet of test cases. The conventional views of static and dynamic analysis are as follows:

All authors are with the David R. Cheriton School of Computer Science, University of Waterloo, Canada.  
E-mail: {a25murali, malfadel, mei.nagappan, meng.xu.cs, cn-sun}@uwaterloo.ca

- Static analysis techniques can spell out the complete semantics of a memory object (from allocation to free) on all paths, but this is not precise (e.g., Memfix [7], Leakfix [8]).
- Dynamic analysis techniques can spell out the precise semantics of a memory object on a particular execution path. This is enough for leak detection but less useful for suggesting a fix covering multiple execution paths (e.g., LeakPoint [11], LeakChaser [13]).

AddressWatcher offers a way to combine the strengths of both views—it allows the semantics of a memory object to be tracked on multiple execution paths, only bounded by the quality of the test suite. In particular, AddressWatcher uses a leak database as a dynamic analysis technique, storing and comparing execution traces of leaks over several test runs. It also complements prior static analysis tools by considering memory leak cases that are unable to be fixed by such static-based analysis, i.e., our approach relies upon test case runs to track execution paths of leaks, and hence, it does not suffer from issues related to “path explosion” problems that are present in static-based approaches. Moreover, our approach addresses the problem of slowdown due to dynamic binary instrumentation in certain dynamic-based approaches (e.g., LeakPoint) by using sanitizer-based and light-weight compile-time instrumentation.

**Contributions.** The key contributions of this paper are as follows:

- 1) We present AddressWatcher, an automated dynamic analysis tool that suggests locations for memory leak fixes in C/C++ programs. AddressWatcher introduces the concept of using shadow memory to tag memory and eventually suggests a bug-fix location. AddressWatcher can suggest multiple free locations for a given leak after considering all relevant execution traces in a leak database.
- 2) We examine the effectiveness of AddressWatcher on a set of 50 memory leak bugs in popular packages, followed by a qualitative analysis. We compare the fix locations suggested by AddressWatcher with Memfix [7].
- 3) We demonstrate the practical relevance of AddressWatcher by submitting 25 PRs to major open-source projects with memory leak issues. Of these, 21 were approved and merged, resolving 5 open GitHub issues [14, 15, 16, 17, 18].
- 4) We develop a prototype tool for AddressWatcher. The tool and benchmarks are made publicly available [19].

## 2 MEMORY LEAKS & SANITIZERS

In this section, we provide an overview of several concepts related to memory errors.

**Shadow memory** is a duplicate region of memory used to mimic the state of actual process memory. It can be used to store any kind of information about the state of process memory [20, 21, 22, 23]. **Compile-time instrumentation** refers to the insertion of appropriate instructions into the application binary during compilation. These instructions can be used to detect the violation of a given property to identify memory errors [24, 25]. **Red zones** are fixed-size

blocks of memory that are safe from modification by a given application. Red zones have been used to detect several memory errors. For example, a common technique to detect buffer overflows is to pad local and global variables with red zone buffers. If a read/write operation happens to a red zone then a buffer overflow has occurred. A common technique to identify the location of buffer overflow is to use compile-time instrumentation to insert checks at suspicious read and write locations. The instrumentation checks if the read/write happens to a red zone memory region. Several tools (e.g., ASAN [21], Dr. Memory [20]) have been proposed to detect such scenarios by using shadow memory to encode the location of inserted red zone regions.

**Memory leak checking** is an example of a memory error whose source can be traced by red zones that store relevant information about the surrounding process memory. Whenever memory is allocated, a red zone region can store metadata such as a thread ID for allocating the memory, the size of allocated memory, and the program stack at which the memory was allocated. Tools such as LSan [26] ensure that when the allocated memory is freed, it is overwritten with a magic value. Before the program termination, LSan detects memory leaks by identifying allocated memory surrounded by red zone regions that have not been overwritten with a magic value. The allocation stack is then retrieved from the red zone as the source of the memory leak.

```

1 void read(int size) {
2     char* p = (char*)malloc(size);
3     fgets(p, size, stdin);
4     if (*p == "\n") {
5         // Path 1: Abrupt return
6         // Leak p
7         return;
8     }
9     // Path 2: Process p
10    // Leak p
11    return;
12 }

```

Listing 1: Leak detection vs leak fixing.

**Leak detection vs fixing.** While memory leak detection and leak fixing are relevant, they require fundamentally different approaches to address them. In the case of static leak detectors such as Saber [27], it is sufficient to track allocated memory being leaked along just *one* path to confirm a leak. For example, in code listing 1, user stdin input is stored in “p”. If input starts with a newline character, then we return abruptly on Path 1 (see line 7), which leaks allocated memory. Otherwise, we proceed to process user input normally on Path 2 (see line 11), which also leaks the memory. A static tool can analyze Path 1 alone to confirm the leak. However, in order to fix the leak, a static technique must analyze *all* paths that leak memory in order to fully deallocate the leak (by inserting frees along both Path 1 and Path 2).

## 3 DESIGN

AddressWatcher is an approach that performs memory tracking to automatically suggest locations where a fix for a memory leak should be placed.

To put the problem more formally: consider a memory allocation at code point  $A$  within a program that is subsequently leaked. Assume that the allocated object is used at different code points  $o_1, o_2, o_3, \dots, o_n$  within the program. A use of an allocated object can refer to either a read or write to the allocated object. Intuitively, the fix to this memory leak should be placed after the last use of the allocated object. Therefore, the core problem AddressWatcher aims to solve is how to automatically identify the last use of a memory object.

On a high level, AddressWatcher relies on dynamic information collected when executing test cases of the target program. To illustrate, assuming the testsuite for this program contains two different testcases  $t_1$  and  $t_2$ . Let us assume that testcase  $t_1$  executes a sequence of these code points  $A, o_2, o_3, o_4$  etc. Similarly the other testcase  $t_2$  can execute a different sequence of code points  $A, o_1, o_5$ . AddressWatcher needs to identify that  $o_4$  and  $o_5$  are the last points where memory related to allocation  $A$  was used and hence needs to be subsequently freed.

AddressWatcher breaks down the challenge of suggesting memory leak fixes into the following (D)esign components:

- (D1) **Instrument relevant code points.** Set up checkpoints at all relevant code points through binary instrumentation, i.e.,  $o_1, o_2, \dots, o_n$ .
- (D2) **Detect leak.** At the end of a given testcase, AddressWatcher must detect that the allocation at  $A$  is not deallocated.
- (D3) **Tag leak.** At a given allocation code point while executing a testcase, AddressWatcher must detect that the allocation was leaked on some other testcase in the past.
- (D4) **Track leak.** Identify when leaked memory is read/written at a given instrumented checkpoint.
- (D5) **Preserve leak and execution trace integrity across testcases.** Store leak information and execution trace of leaks in a database in a way that preserves integrity across multiple testcases.
- (D6) **Suggest fix location.** Compare all the different execution traces (eg. execution traces for testcases  $t_1$  and  $t_2$ ) to identify multiple last use points, after which a fix is suggested.

Each of the design components (mentioned above) represents unique challenges. In Section 4, we discuss how each of these challenges is handled.

## 4 PROPOSED APPROACH

In the previous section, we discussed the design challenges that AddressWatcher aims to tackle. In this section, we discuss how each unique challenge is handled in the AddressWatcher framework. Figure 1 shows an overview of our approach. The Figure shows a code program that suffers from a memory leak on line 1. Our goal is to suggest a location where a fix to the memory leak should be added, i.e., the approach aims to suggest a code line for adding a deallocation statement. Our approach goes through four main steps to achieve the goal, (1) Adding a leak checker and instrumenting program binary; (2) Identifying memory leaks; (3) Tagging and tracking execution paths of memory leaks; (4) Suggesting fix location for memory leaks. These

steps outline a concrete implementation to achieve each of the design components described in the previous section.

Next, we describe each step in detail using the code example shown in Figure 1. We explain how the approach goes through the four steps to suggest a fix location after line 3 for the memory leak that occurs in the code example.

**Step 1: Adding a leak checker and instrumenting program binary.** The first goal of this step is to obtain an instrumented binary of a given program. The compiler adds AddressWatcher instrumentation before each read and write operation in the program. As shown in Figure 1, AddressWatcher adds some code before lines 2 and 3, since these lines contain read/write operations of program memory. The instrumented binary is an essential artifact of the approach as it helps for tracking memory leaks (more on this in later steps). The AddressWatcher instrumentation behaves as a checkpoint while tracking reads and writes of memory objects. This enforces the design component (D1) described in the previous section. The other goal of this instrumentation step is to insert sanity-checking logic for detecting memory leaks. The output of this step is an instrumented binary of the given program.

**Step 2: Identifying memory leaks.** In the previous step, we are able to obtain an instrumented binary of the program. In this step, we use the instrumented binary and identify memory leak objects (e.g., code points that have malloc, calloc, and realloc allocations without certain deallocations in the program). To achieve this step, we execute the instrumented binary which essentially runs the existing test cases. Just before the program is terminated while running the tests, the leak checker module (which was instrumented in step 1) is invoked to perform heap analysis to detect leaked objects (as described in Section 2). For the running example in Figure 1, the leak checker identifies that there is a leaked object originating from the calloc function on line 1. The leak checker detects leaks at the end of the given testcase which accomplishes the design step (D2). In fact, the checker additionally stores the allocation stack traces of the leaks in a database, which we call a leak database. The allocation stack trace is the program stack when the leaked object is allocated. Initially, the leak database is empty, however, after the first program execution, the leak database gets populated with the detected leaks. This is one essential part of achieving the design component (D5). Subsequent testcases will then read from the leak database to match allocations with leaks from testcases in the past. Such a database aims to help us for profiling and tracking the execution paths of each leaked object, as we describe in the next step.

**Step 3: Tagging and tracking execution paths of memory leaks.** So far we are able to only identify leaked objects of the program (stored in the leak database). In this step, our goal is to obtain the execution paths of the previously identified leaked objects. Obtaining the execution path of the identified leak is important to deduce the fix location in the next step. The execution path of a leaked object is a list of stacks that represent code points that the leaked object passes through. To obtain these stacks, we re-execute the program binary, and perform two tasks: 1) tagging leaks; and 2) tracking

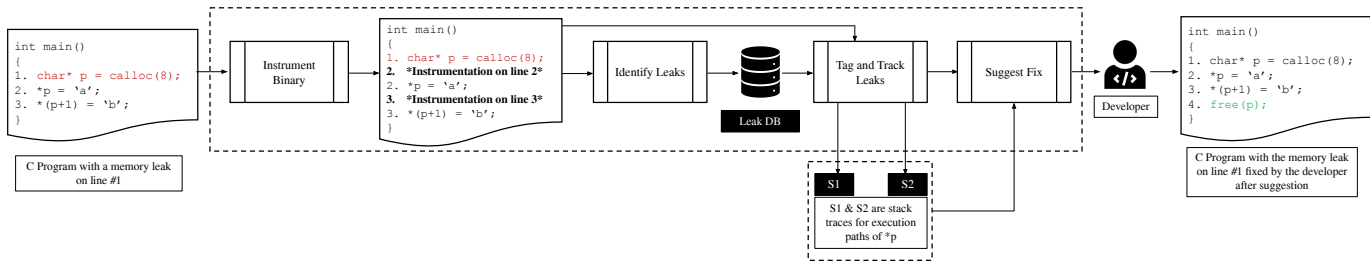


Fig. 1: An overview of our approach for suggesting a location of a memory leak fix.

leaks.

**Tagging leaks.** To enable tracking leaked objects during the program execution, we first need to distinguish leaked from non-leaked memory objects, i.e., we need to tag leaked objects to be tracked during the execution. To perform tagging, we examine if each allocation is a memory leak in a previous execution, by obtaining its allocation stack and then comparing it to the stacks stored in the leak database. If there is a match with any stack in our leak database, we add a special value (i.e., a tag) to the leaks' shadow memory. Given the code program in Figure 1, once our program execution reaches line 1 in the code, a check happens to examine if the allocation is a memory leak. Given that there is a match in the leak database, we tag the leaked object on line 1, i.e., a special value that corresponds to the leaked object `p` is added to its shadow memory. This fulfills design component (D3).

**Tracking leaks.** Once the leaked object is tagged, we want to track its execution path, i.e., we track all code points that the leaked object passes through. For example, the code shown in Figure 1 contains `calloc` function on line 1 for variable `p` (which we already know is a memory leak). Subsequent lines that write to `*p` (i.e., lines 2 & 3) are a set of lines that are part of the execution path of the leaked object. Hence, we need to collect the stack traces of both lines 2 and 3. To do so, we utilize the instrumentation (added in step 1) as guard checkpoints surrounding every read and write operations on memory, in order to check whether the read/write happens on a tagged leaked object, by checking the tag value of the object in the shadow memory. If the shadow memory of the object contains the special value, we record the current program stack in the execution path of the leak. Since both conditions are satisfied, i.e., line 2 and line 3 are memory writes to variable `p` and `p` is already tagged in the shadow memory, we record the stacks related to both lines. Overall, this step implements design component (D4) discussed in previous section. Finally we store the stacks as part of the execution path of `p` in the leak database. This is the final crucial part of design component (D5). The output of this step is a list of stacks that represent execution paths of a memory leak, e.g., an execution profile consisting of stacks `S1` and `S2` (corresponding to lines 2 and 3) is the output after applying this step on our given program, as shown in Figure 1. Note that we provide more details about the special value of shadow memory and tracking process in Technical Challenges Section 5.

**Step 4: Suggesting fix location for a memory leak.** In the previous steps, we are able to profile and obtain a set of

different execution paths (list of stack traces) along which memory is leaked. In this step, we utilize the execution paths to suggest fix locations of a memory leak. Ideally, the last code point in the tracked execution path of a memory leak allocation is considered to be a fix location. For example, in the code shown in Figure 1, to fix the memory leak on line 1, a developer needs to add a deallocation statement after line 3, as this is the latest line in which the leaked object `p` is being used. To identify such a fix location, we identify the last code point on our tracked execution profile. The execution profile for the leak in our example consists of stacks `S1` followed by `S2`. Therefore the last code point in this execution profile is the stack `S2`. Our approach aims to suggest the stack of the last use point as a fix to the leak, and hence, a developer can benefit from our solution by manually adding a `free` statement at the suggested point, i.e., the developer needs to deallocate the leak immediately after line 3. The solution is further refined over several test runs as more execution paths are uncovered, i.e., we find the last use point over all of the execution traces stored in the leak database. Note that if the program contains multiple `malloc` functions, our approach still tackles them because we define a separate execution profile for each individual leaked object. `AddressWatcher` also handles scenarios where multiple `free`s are required for a given memory leak. A detailed explanation of the `AddressWatcher` algorithm that handles multiple deallocations is described in Technical Challenges Section 5. This step enables the realization of the design component (D6).

## 5 TECHNICAL CHALLENGES

In the previous section, we provide a high-level description of how `AddressWatcher` works to suggest a memory leak fix. In this section, we describe specific implementation details of different components of the approach to provide insights into the technicalities of our work. We build prototype for `AddressWatcher` atop `ASAN` [21] and `LSAN` [26] runtime libraries within `gcc` compiler.

**Detecting leaks.** We use a sanitizer for detecting memory leaks - `LSAN` [26]. During compile time, we compile with `LSAN` option, which add a leak checker module to identify leaks in the program.

**Implementing leaks database.** A key component in our approach is the leak database (step 2, Section 4). To implement the leak database, we create a text file to store allocation stacks of the leaks and their execution trace. Note that we create a unique leak database for every binary, i.e.,



the database name is a combination of the instrumented binary name and the instrumented binary directory path. We also use a special directory to store the leak database. This directory has read and write permissions only for the given user running the binary.

**Recompilation constraints.** The design of AddressWatcher also takes into account the case when a program binary gets recompiled. For example, for the code example shown in Figure 1, AddressWatcher suggests adding a fix after line 3. Let us assume that a developer agrees with the suggestion and inserts a free statement in line 4. In this case, if a recompilation happens after, AddressWatcher must stop tracking the memory leak allocation at line 1 because the leak has been fixed. However, the leak database still has a stack stored in it, suggesting that allocation at line 1 is a leak. Hence, recompilation renders all information in the leak database useless. Therefore, we label each leak database with the compile time of the corresponding binary, and if a recompilation is detected, AddressWatcher flushes the corresponding binary’s leak database.

**Implementing leak tagging.** AddressWatcher uses shadow memory to encode information on whether a region of memory is tagged as a leak, i.e., we override allocation functions such as malloc/calloc/realloc, so that AddressWatcher can check if the allocation stack trace of the object belongs to the leak database. If so, AddressWatcher tags the allocated object by assigning a special value to its shadow memory before finally allocating the memory. This special value added to the shadow memory corresponds to setting the higher-order word (16 bits) of the shadow memory to hexadecimal value `0xe`. We note that unlike previous approaches [20, 21, 22, 23] that utilize shadow memory to detect memory errors, AddressWatcher utilizes shadow memory to tag leaks on different executions and to finally suggest leak fixes.

**Implementing leak tracking.** AddressWatcher utilizes instrumentation to perform leak tracking. This instrumentation is a modification of ASAN instrumentation [21]. Before reading and writing to allocated variables of the program, ASAN inserts instrumentation. The ASAN instrumentation is shown in Listing 2. Specifically, in lines 1 and 2, the shadow value of variable `Var` is retrieved by accessing shadow memory. In line 3, the instrumentation checks the shadow value. If it is not 0, `Report` function is called on line 4. We use the function `Report` to track execution paths of leaked objects (step 3, Section 4). For tagged leaked objects, the special shadow value is always non-zero (as we modify the upper word of shadow value when tagging leaks). Hence, the condition shown on line 3 always succeeds and `Report` is invoked. If the shadow value contains the special value tag, AddressWatcher adds the current program stack to execution path of the tagged leaked object. Otherwise (If the special value tag is not observed), AddressWatcher continues execution normally.

```

1 ShadowAddr = ShadowMap(&Var);
2 ShadowValue = *ShadowAddr;
3 if (ShadowValue != 0)
4     Report (&Var);

```

Listing 2: ASAN instrumentation.

TABLE 1: Execution trace of example testcases considered.  $A$  is allocation code point for leaked object.  $o_1$ ,  $o_2$  and  $o_3$  are code points where the leaked object is used.

Testcase	Code points executed
t1	$A, o_1$
t2	$A, o_1, o_2$
t3	$A, o_1, o_3$

**Multiple free locations.** There can be multiple locations for free statements required to fix a given memory leak. AddressWatcher resolves this by analyzing all the tracked execution paths of the memory leak from the leak database. We illustrate this with an example below. For example, consider a case where an object is allocated at code point  $A$  and  $o_1, o_2, o_3$  are different code points where the memory allocation is used. Let us assume there are three testcases t1, t2, and t3 which leaks the same object on different execution paths. We describe the execution trace of each testcase as a list of code points executed by the testcase in Table 1. AddressWatcher analyzes each execution trace and filters those execution traces which are a subsequence of another execution trace. A subsequence of an execution trace  $E$  is the resulting execution trace when one or more code points are deleted within  $E$ . For example the execution trace of t1 is a subsequence of the execution trace of t2 (i.e. code point  $o_2$  must be deleted from execution trace of t2 to obtain execution trace of t1). We filter execution trace of t1 because if a free is inserted at its last code point (i.e.  $o_1$ ) it will result in a use-after-free vulnerability when testcase t2 is executed. AddressWatcher then obtains the last code point within the remaining execution traces which are points where the developer is suggested to add the deallocation statements. We note that as the test suite covers more leaked paths, AddressWatcher refines its fixes by storing relevant execution traces in the leak database for analysis.

**Custom API deallocations.** AddressWatcher only suggests the location of the last use point of leaked memory. The developer must identify the pointer to the allocated memory from this last use point and manually insert the deallocation statement thereafter. The developer can deallocate this leaked memory at the suggested location in their own desired way using custom APIs. We note that these custom APIs still fundamentally deallocate memory using `free` function.

## 6 EVALUATION

In this section, we evaluate our proposed approach. In particular, we introduce our evaluation dataset in Section 6.1 and present how fixes suggested by AddressWatcher compare with manual fixes from developers in Section 6.2. Next, we conduct a comparison with Memfix tool, the state of the art approach for statically fixing memory leaks, in Section 6.3. We then present the efficiency evaluation of AddressWatcher in Section 6.4. Finally, we demonstrate the practicality of AddressWatcher by submitting 25 PRs across 12 diverse open-source software. We discuss these PRs in depth in Section 6.5.

## 6.1 Benchmark

We evaluate AddressWatcher over real world memory leaks that have been fixed by developers in open-source projects. In particular, we choose to evaluate AddressWatcher over the same benchmark examined by Memfix, a static-based approach for fixing memory leaks in C/C++ programs [7]. The benchmark used by Memfix includes memory leak bugs in five open source packages, namely, `openssh`, `binutils`, `tmux`, `openssl` and `git` [28, 29, 30, 31, 32].

The benchmark in our data comprises 50 fixes of memory leak bugs distributed across the five packages. There are 10 bug fixes in each of the following repositories - `binutils`, `tmux`, `openssh-portable`, `openssl` and `git`. To verify that these bugs are memory leak fixes in the program, we manually analysed them by looking at the corresponding GIT commit, in order to ensure that it discusses fixing a memory leak bug through information in the commit message and the discussion of the corresponding pull-request.

The ground truth for the bug fixes in our benchmark are locations (line numbers) where a developer inserted a `free` statement for the corresponding memory allocation. To perform our evaluation, we compare this ground truth with the solution suggested by AddressWatcher. If AddressWatcher suggests a fix to be added at the same location as the one inserted by the developer (ground truth), we consider that AddressWatcher successfully fixed the memory leak bug. In contrast, if AddressWatcher suggests a fix that has a different location from the ground truth, we consider AddressWatcher to fail fixing the bug.

## 6.2 AddressWatcher Effectiveness

Table 2 presents the distribution of 50 bug fixing commits across the three studied packages in our dataset. For each fixing commit, Table 2 shows whether AddressWatcher fixes the corresponding bug. Overall, AddressWatcher fixes a total of 23 bugs (out of 50 bugs) with an accuracy of 46%. Breaking down the fixes per package, AddressWatcher fixes 6 bugs in `binutils`, 6 bugs in `tmux`, 3 in `openssh`, 3 in `openssl` and 5 in `git` covering a total of 23 bugs. That said, AddressWatcher fails to fix the remaining 27 bugs.

We manually look at each one of the 27 bugs to understand the context of the affected code and the reason why AddressWatcher could not suggest the same fix as the one developers add in the code. Table 2 summarizes the reasons as to why AddressWatcher failed to fix a given memory leak bug. Below, we provide more details about each reason.

**Error paths (20 cases).** The most common reason of AddressWatcher failing to provide a correct fix is when the program contains error paths. An error path refers to a program path where an abrupt return happens under abnormal situations (incorrect arguments, low memory, etc.). Listing 3 shows a real example of such a case for a bug in `binutils` [33]. In the Listing, the error path could be triggered in case the user does not supply the correct arguments to the program. The allocation on line 1 is leaked only on the error path (when `goto FAIL` is executed on line 2) and not on the non-error path (for the non-error path, there is already a deallocation on line 4). Hence, a

```

1  struct btrace {
2      char* p;
3      char* q;
4  }
5  void btrace_alloc (struct* X) {
6      X = malloc(sizeof(X));
7      X->p = malloc(10);
8      X->q = malloc(10);
9  }
10 void btrace_clear (struct* X) {
11     // Developer adds free(X->p) here
12     free(X->q);
13     free(X);
14 }
15 int main () {
16     struct btrace* X;
17     btrace_alloc(X); // Allocate X,p,q
18     doSomething1(X->p); // Use X->p
19     // AddressWatcher suggests fix for X->p here
20     doSomething2(X->q); // Use X->q
21     btrace_clear(X); // Deallocate X,q
22     return 0;
23 }

```

Listing 4: Code snippet from `binutils` showing AddressWatcher failure due to code organization [34].

developer would insert the fix in the error path before the `exit` keyword on line 8. In such cases, AddressWatcher fails to provide the same fix as the developer’s solution. This is because there is no read/write operation to the leaked object in the error handling routine, and hence, our approach is not able to track the leaked object. In such cases, AddressWatcher suggests a fix immediately after the allocation on line 1.

```

1  char *p = malloc(10); // AddressWatcher suggests
   fix after allocation
2  if(argv == NULL) goto FAIL; // Memory leak
3      *p = 'a' // Non-error path
4  free(p);
5  return 0;
6  FAIL:
7  // Error path
8  // Developer inserts fix here
9  exit(1);

```

Listing 3: Code snippet from `binutils` showing AddressWatcher failure due to an error path [33].

**Weak test suite (5 cases).** Other cases where AddressWatcher could not suggest an accurate fix is when the program test coverage is weak or insufficient. This is because AddressWatcher relies upon the test suite to track leaks on new paths. That said, with a better testsuite coverage, AddressWatcher is still able to suggest the correct fix for these cases.

**Code organization (2 cases).** Code organization refers to developer’s decisions in structuring the code in a certain way to promote general code reliability and adaptability to future changes. This can lead developers to provide fixes that are not always immediately after the last use of the allocated object. Such a case is seen in a bug that affects the package `binutils` [34], which is illustrated in Listing 4. As the Listing shows, there is a memory leak originating from

TABLE 2: List of 50 bug fixing commits in our benchmark, per package. For each commit, we present: 1) whether AddressWatcher can fix the bug, 2) the reason AddressWatcher fails to fix the bug, 3) whether Memfix can fix the bug.

Repository	Leak No.	Fixed by AW (✓/×)	AW Failure reason	Fixed by MF? (✓/×)
binutils	1	✓	-	✓
	2	✓	-	✓
	3	✓	-	✓
	4	✓	-	✓
	5	✓	-	×
	6	✓	-	×
	7	×	Error path	×
	8	×	Code Organization	×
	9	×	Code Organization	×
	10	×	Error Path	×
tmux	1	✓	-	✓
	2	✓	-	✓
	3	✓	-	✓
	4	✓	-	✓
	5	✓	-	×
	6	✓	-	×
	7	×	Error Path	✓
	8	×	Error Path	×
	9	×	Error Path	×
	10	×	Error Path	×
openssh-portable	1	✓	-	✓
	2	✓	-	×
	3	✓	-	×
	4	×	Error Path	✓
	5	×	Error Path	✓
	6	×	Error Path	✓
	7	×	Error Path	✓
	8	×	Weak test suite	✓
	9	×	Error Path	×
	10	×	Weak test suite	×
openssl	1	✓	-	✓
	2	✓	-	×
	3	✓	-	×
	4	×	Error Path	✓
	5	×	Error Path	✓
	6	×	Error Path	✓
	7	×	Error Path	✓
	8	×	Error Path	×
	9	×	Error Path	×
	10	×	Weak test suite	×
git	1	✓	-	✓
	2	✓	-	×
	3	✓	-	×
	4	✓	-	×
	5	✓	-	×
	6	×	Weak test suite	×
	7	×	Weak test suite	×
	8	×	Error Path	×
	9	×	Error Path	×
	10	×	Error Path	×

allocation of variable `p` within struct `X` of type `btrace`. The developer constructs two separate routines for allocation and deallocation of the struct `X` and its inner variables, one is called `btrace_alloc` (on line 5) which is used for allocation, and the other is called `btrace_clear` (on line 10) for deallocation. `X->p` is allocated in `btrace_alloc` and its last use occur in routine `doSomething1`. Hence, AddressWatcher suggests the last use point after routine `doSomething1` as the fix. However, the developer inserts the fix on line 11 in `btrace_clear` before the struct `X`

TABLE 3: Memory leak fixes by AddressWatcher & Memfix, per package.

Repository name	Total bugs	# Fixes by AddressWatcher	# Fixes by Memfix
binutils	10	6	4
tmux	10	6	5
openssh-portable	10	3	6
openssl	10	3	5
git	10	5	1
Total	50	23	21

itself is freed. This could have been done for general code reusability reasons for future commits. We count this as a negative case for AddressWatcher because we are comparing with the ground truth being the location where the developer inserts the fix. However, we should note that the solution suggested by AddressWatcher is more optimal (freeing memory earlier) than the developer’s fix.

### 6.3 Dynamic (AddressWatcher) Fixing Compared to Static (Memfix) Fixing

In this section, we compare AddressWatcher to Memfix, the *state-of-the-art* approach for statically fixing memory leaks [7]. The goal of our comparison with a static-based analysis approach is to expose the gap between static and dynamic analysis for fixing memory leak bugs.

Memfix attempts to fix memory leaks by identifying a set of free statements that deallocates all objects without causing double-free or use-after-free. It utilizes the insight that identifying set of deallocation statements corresponds to an exact cover problem on a variant of typestate static analysis. It uses a SAT solver to solve the exact cover problem [12]. Then, all frees that are present in these paths are removed, and new free statements are added.

We compare AddressWatcher to Memfix over the same packages examined in the benchmark. We set up a virtual machine provided by Memfix and run Memfix with its default configuration over the bugs in the benchmark [35]. We run Memfix without any time constraints and compare results with AddressWatcher.

Table 3 shows the number of memory leak fixes by both Memfix and AddressWatcher, per package repository. From the Table, we can see that, of the 50 bugs examined in our benchmark, AddressWatcher fixes 23 bugs while Memfix fixes 21 bugs.

We now aim to understand the intersection of memory leak fixes introduced by AddressWatcher and Memfix, i.e., bugs independently and jointly fixed by both tools. We outline the distribution of memory leak fixes in Figure 2. The figure shows that AddressWatcher fixes 12 bugs independently of Memfix, i.e., Memfix fails to suggest fixes for these bugs. Also, we can see that there are 10 bugs fixed by Memfix independently of AddressWatcher. Both AddressWatcher and Memfix jointly provide proper fixes for 11 bugs. Still, 17 bugs in the examined repositories were not fixed by both AddressWatcher and Memfix. Next, we provide further investigation into those bugs which are not fixed by atleast one approach (static or dynamic). We describe these in the following three scenarios (S1 – S3):

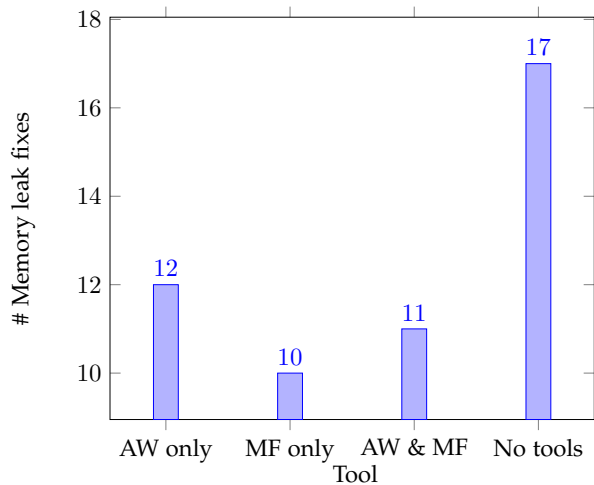


Fig. 2: Distribution of memory leak fixes by AddressWatcher (AW) and Memfix (MF).

### S1. Memory leak bugs fixed by AddressWatcher only.

As shown previously, AddressWatcher provides 12 fixes independently of Memfix. Among them, Memfix fails to generate and solve constraints for its SAT solver in 6 cases due to the complex nature of these interprocedural leaks. Memfix also fails in 3 cases where leaked memory is reallocated (with realloc). Finally, Memfix fails in the remaining 2 cases where recursion is not supported. Additionally, in one case, the Memfix application crashes without providing a reason for the crash. In contrast, AddressWatcher succeeds in such cases because it is a dynamic approach where it relies on a test suite to uncover all leaked paths.

### S2. Memory leak bugs fixed by Memfix only.

To understand the nature of the 10 bugs fixed by Memfix independently of AddressWatcher, we manually look at these cases, trying to understand the nature of each case. We find that most of these bugs are related to error path cases (9 cases). Another failure case is due to a weak test suite not covering the leaked path. We previously describe the context of all such cases where AddressWatcher fails in Section 6.2.

### S3. Memory leak bugs not fixed by both AddressWatcher and Memfix.

For the 17 cases that both approaches fail to provide a fix, our manual investigation shows that the majority of the bugs that AddressWatcher could not fix are due to error paths (11 cases). There are 4 memory leaks where AddressWatcher fails to fix due to a weak test suite, and the remaining 2 cases are due to code organization issues. Memfix encounters issues in specific scenarios: it fails when dealing with reallocated memory (10 cases), cannot generate constraints for its SAT solver in certain complex interprocedural leaks (6 cases), or when dealing with function pointers (1 case).

To sum up, our comparison demonstrates that AddressWatcher complements Memfix by tackling fixes along non-error paths, where Memfix errors out. Moreover, note that

TABLE 4: Efficiency of AddressWatcher (AW).

Repository	% AW overhead	AW avg run time (s)	AW max run time (s)
binutils	178.24	0.53	0.62
tmux	160.69	0.51	0.54
openssh-portable	142.75	0.52	0.75
openssl	147.32	0.50	0.58
git	171.36	0.48	0.60

Memfix does not support recursion, function pointers and memory reallocation in their static analysis, which is not the case in dynamic approaches like AddressWatcher. These tools are, hence, complementary in nature and can be used together to fix 33 real-world leaks in our benchmark.

## 6.4 AddressWatcher Efficiency

In this section, we evaluate the efficiency of AddressWatcher. AddressWatcher utilizes LSAN to detect leaks. Then, in subsequent testcases, it profiles those leaks and suggests final fix locations. Therefore, for Addresswatcher to provide the best solution, each testcase must be run twice. This is the runtime with AddressWatcher instrumentation. In order to calculate AddressWatcher overhead, we must first establish the time taken to run those testcases without any profiling instrumentation or LSAN. Hence, our baseline is the runtime of the binary with testcases run twice without any instrumentation. Then, the overhead is calculated as follows:  $((T_I - T_{WI})/T_{WI}) * 100$  where  $T_I$  is the time taken to execute testcases twice with instrumentation and  $T_{WI}$  is the time taken to execute testcases twice without instrumentation. The result of this analysis is shown in Table 4.

Table 4 indicates the average runtime required by AddressWatcher to suggest fixes in a repository. The table also shows the maximum runtime to suggest the leak fix in that repository. Overall AddressWatcher has a performance overhead of 142-178% over the baseline. This clearly shows the efficiency of AddressWatcher in suggesting memory leak fixes. Additionally, AddressWatcher is a debugging tool, not a runtime checker, and is not intended to be deployed on production servers. Therefore, it is most useful when the code is in the testing phase.

## 6.5 Practicality of AddressWatcher in Open Source

In this section, we demonstrate the practicality and relevance of AddressWatcher to the open-source community. We identified 12 prominent C repositories with memory leak problems using the Github search functionality. We then used the Leaksanitizer tool to detect various memory leaks. We then submitted 25 Pull Requests to these prominent open-source projects that are affected by memory leak bugs. We submitted these PRs by inserting deallocation statements at the locations specified by the AddressWatcher suggestions.

Overall, among these 25 PRs, 21 were merged and 4 are pending approval. The merged PRs led to 5 open github issues on memory leaks being resolved [14, 15, 16, 17, 18].



One of our merged patches was so critical that it resulted in a new version of the calc software (calc v2.15.0.6) being released [36]. Additionally, 3 of our PRs even triggered lively discussion on the coturn repository on how to improve memory safety in the future by upgrading to C++17 [37]. Below, we provide insights into the context of a subset of these PRs, highlighting their importance and the impact they have on the analyzed projects:

```

1 sockfd = openconn(server, port);
2 free(server); // AddressWatcher fix
3 server = do_query(sockfd, query_string);

```

Listing 5: whois PR showing AddressWatcher leak fix [38].

```

1 if (getsockopt(fd, IPPROTO_TCP, TCP_CONGESTION,
2 buf, &buflen) == 0) {
3     return h2o_iovec_init(buf, strlen(buf));
4 }
5 free(buf); // AddressWatcher fix

```

Listing 6: h2o PR showing AddressWatcher leak fix [39].

```

1 // Displayed if appending to trash fails when
2 syncing or closing a mailbox
3 if (mutt_append_message(m_trash, m, e, NULL,
4 MUTT_CM_NO_FLAGS, CH_NO_FLAGS) == -1)
5 {
6     mx_mbox_close(m_trash);
7     mailbox_free(&m_trash); // AW fix
8     return -1;
9 }

```

Listing 7: neomutt PR showing AddressWatcher fix [40].

- 1) Calc (1 PR merged): Calc [41] is an interactive calculator that can be easily programmed for difficult or long calculations. It has been used before to calculate the largest known non-Mersenne prime:  $391,581 * 2^{216,193} - 1$ . Calc is now being used to calculate other large primes over years of execution. We merged a PR in this repository [42] and were given credit for fixing a “long-standing memory leak” on their discussion forum [36] and CHANGES document [43]. The repository owners noted that the merged fix addresses a memory leak in calc that previously caused crashes due to gradual memory leak buildup over years of execution, hindering the discovery of larger primes. Additionally, a new version of calc was immediately rolled out (calc v2.15.0.6) after merging the PR indicating the high impact of the fix [36].
- 2) Radare2 (10 PRs merged): Radare is a library used to ease binary reverse engineering tasks and binary exploitation [44]. We submitted 10 PRs fixing leaks detected while analyzing “ls” binary and by running the visual mode of radare [45, 46, 47, 48, 49, 50, 51, 52, 53, 54]. Three of these PRs fixed leaks across multiple locations as well [47, 48, 49]. All 10 were merged. In one PR, the author was even impressed with a fix saying “whoa nice spot!” [54].
- 3) CoTurn (3 PRs merged): Coturn is an implementation of a VoIP media traffic NAT traversal server and gateway [55]. We submitted 3 PRs to this repository and all 3 were merged [56, 57, 58]. The PRs that we submitted initiated a long discussion on how to migrate

project code to C++17 to improve memory safety using destructors [37]. Later after submitting the PRs, the first author was approached by mail for a consultancy position.

- 4) Net-snmp (2 PRs merged): Net-snmp is a widely used protocol for monitoring the health of network equipment [59]. We fixed 2 leaks identified by fuzzers when the ‘-a’ argument of net-snmp is used incorrectly [60, 61]. Both PRs were merged resolving an open issue [16].
- 5) Neomutt (1 PR merged): NeoMutt is a command line mail reader [62]. We submitted a PR fixing a memory leak when certain mail is added to trash [40], as shown in Listing 7. In this case, the trash mailbox is closed, but the memory associated with it is not freed. Address-Watcher is able to locate the line that closes the mailbox as the last use location and suggests a free after this line. The PR fixing the leak was merged and the authors were congratulated for a “good spot”.
- 6) Whois (1 PR merged): Whois is a UNIX command-line utility used to make WHOIS protocol queries [63]. In the repository, a pointer to allocated memory `server` is reassigned without being freed, as shown in Listing 5. AddressWatcher is able to detect this last line where the leaked memory is used and suggests the free after this line. We submitted a PR fixing the leak that is merged [38] and resolved another issue [17].
- 7) h2o (1 PR merged): h2o is a really fast HTTP server [64]. In Listing 6, a leak occurs when allocated memory in `buf` is used to change socket options through `getsockopt`. If setting socket options fails, the allocated memory is never used and never freed. AddressWatcher identifies last use location to be the `getsockopt` call. We submitted a PR that frees the memory after the branch condition, which has been merged [39] and resolved another issue [15].
- 8) Iniparser (1 PR merged): This library offers parsing of ini files from C [65]. We submitted a PR [66] which project owners mentioned to be more comprehensive than another PR submitted by a different user [67]. Our PR was merged in the repository leading to an open issue being resolved [14].
- 9) NanoNNG (1 PR merged): NanoNNG solves common recurring messaging problems for IoT devices, such as RPC-style request/reply [68]. We merged a PR that fixed a memory leak that occurred on reading configuration files with repeating JSON fields [69]. This also led to an open issue being closed [18].

We also submitted 4 PRs in 3 other repositories. We submitted 2 PRs in yasm repository [70, 71], 1 PR in shc repository [72], and 1 PR in hackem repository [73]. However, these PRs are still under review and are yet to be merged.

## 7 DISCUSSION

This section discusses how certain aspects impact the performance of AddressWatcher when suggesting fix locations.

**Code coverage.** AddressWatcher fundamentally relies on a test suite to provide a fix for memory leaks. Hence, if the test suite does not cover all paths of a leaked object, the

suggested solution might not be optimal. For example, a leaked object may be used through a path that is not covered by the test suite and such a path could be located in a point that is deeper in the program flow than the point we suggest for the fix. Consequently, the suggested fix may lead to a use-after-free if the test coverage is poor. We note that this is a limitation of all existing dynamic analysis techniques [10, 11, 13]. Possible future work to mitigate this in AddressWatcher would be to fuzz the program to obtain a high-quality test suite. This is an orthogonal problem that will further improve the results of AddressWatcher. AddressWatcher is the framework that utilizes these test cases to obtain memory leak fixes.

**Multi-threading support.** In the case of a multi-thread program, each thread could lead down a different execution path for a given leak. Consider the case where a memory leak has two different execution paths. Thread 1 explores the first execution path and thread 2 explores the second execution path. That is, each thread can lead to a different solution for the same leak. To mitigate this, we implement a concurrency lock to prevent threads from writing to the database simultaneously. In the case that thread 1 completes before thread 2, thread 1 first writes its solution to the leak database. When thread 2 is about to die, it reads the previous solution in the leak database and compares it with its own solution using the comparison operator. Based on the comparison outcome, thread 2 will store the best solution, i.e., the solution that represents a later point in the program path will be stored in the leak database.

## 8 RELATED WORK

Two lines of work are closely related to AddressWatcher: (1) techniques for memory leak detection, and (2) techniques for memory leak fixing. In the following, we discuss the related work and reflect on how the work compares with our work.

### 8.1 Approaches for Memory Leak Detection

**Static approaches.** A plethora of recent work focused on detecting memory leaks statically [27, 74, 75, 76, 77, 78]. Static analysis approaches such as Smoke [78] suffers from imprecision in detecting bugs. This can be due to approximations in underlying pointer analysis, lack of library specifications, infeasible paths due to complex arithmetic in branch conditions, and other cases such as recursion, function pointers, etc. Therefore, the common problem of most static approaches is that they incur a high rate of false positives in detecting bugs, and this translates to limitations in bug fixing.

**Dynamic approaches.** Dynamic approaches for memory leak detection have been discussed broadly. For example, LeakSanitizer (LSAN) is a tool by Google that performs dynamic analysis for detecting memory leaks [26]. When memory is freed, a magic value is written into the memory. When the program exits, the heap is checked for memory leaks, i.e., by identifying allocated memory that has not been overwritten with a magic value. Nethercote et al. in their framework Valgrind [79] presents a virtual architecture that captures all calls to memory allocation and deallocation

by the program. When the program exits, Valgrind checks whether a memory allocation has been freed or not. If the allocation is not freed and it is also not reachable from stack and global variables, then it is considered to be a leak. The approach suffers from performance overhead due to the synthetic execution, i.e., checking every memory access.

Our proposed approach, AddressWatcher, suggests memory leak fixes, and is different from static and dynamic approaches for *detecting* memory leaks.

### 8.2 Approaches for Memory Leak Fixing

**Static approaches.** Some prominent techniques have been proposed to statically suggest fixes for memory leaks [7, 8]. For example, Memfix by Lee et al. [7] is a static approach to fixing memory deallocation errors, including memory leaks. First, Memfix identifies all program paths of a leaked object. Then, the bug fix is modeled as an Exact Cover problem where minimum frees must be placed to plug all the leaked paths. A solution to the exact cover problem is generated using a SAT solver. Then, all generated deallocation statements are inserted along the program paths. However, Memfix is not precise and cannot track program paths in the presence of function pointers and recursion. Memfix often errors out when the number of program paths explodes and the SAT solver is unable to generate a solution.

Another static-based approach to fixing memory leaks is Leakfix [8]. Leakfix performs pointer analysis on the whole program. Each procedure is classified into 3 types: those that allocate, deallocate, or use a given memory allocation. It then abstracts the program into a Control Flow Graph (CFG) where every node is a procedure classified into above three types. Then, the task of finding the correct deallocation is equivalent to finding edges in the graph that meet a set of criteria. These are program points where allocated memory is still in scope, but will never be used thereafter in the graph. The precision of the approach is limited to the efficacy of pointer analysis techniques used for the classification of procedures (e.g., DSA [80].) Such techniques are employed to create the graph, but they face limitations. For example, when a pointer array contains two different allocated pointers, DSA cannot distinguish between them.

Our proposed approach, AddressWatcher, is complementary to such static-based analysis approaches. For example, we compare AddressWatcher with Memfix using a dataset of 50 memory leak bugs. We find that AddressWatcher fixes 23 bugs related to non-error paths where Memfix errors out.

**Dynamic approaches.** Several dynamic-based techniques have been proposed to suggest different forms of leak fixes [10, 11, 13]. Yu et al., in their tool DEF\_LEAK, propose a dynamic symbolic execution approach to expose memory leaks occurring in all execution paths [10]. In their approach, the program to be analyzed is instrumented before execution. Dynamic symbolic execution is a technique of analyzing the program to determine inputs that would cause certain parts of the program to execute. This technique is employed to discover as many execution paths as possible. The approach suffers from inaccuracy when it encounters large programs whose inputs are hard to be symbolized in dynamic symbolic execution. AddressWatcher, on the

other hand, relies on predefined test cases to suggest fixes, and hence, the size of the program or nature of input will not affect the accuracy of the suggested solution. Instead, AddressWatcher relies on the completeness of the test suite.

The work most close to ours is proposed by Clause et al. where they proposed LeakPoint [11]. LeakPoint is a dynamic analysis framework that performs taint propagation on pointers to detect leaked objects, in order to identify last-use sites of the objects and suggest candidate sites for fixing them. In fact, Leakpoint uses Valgrind infrastructure for implementing taint propagation and dynamic binary instrumentation, which brings a performance overhead of 100-300 times the base program [11]. We showed in Section 6.4 that AddressWatcher has a performance overhead of 2.42-2.78 times the base program. Unfortunately, we were not able to find a public implementation of all these dynamic tools, such as LeakPoint or DEF\_LEAK, which makes it infeasible to directly compare to AddressWatcher. AddressWatcher, on the other hand, is open source with a publicly available docker [19].

## 9 CONCLUSION AND FUTURE WORK

In this work, we present AddressWatcher, a new framework for fixing memory leak bugs. Previous static analysis approaches attempt to trace the complete semantics of memory objects on all leaked paths with imprecision. On the other hand, dynamic approaches attempt to profile the memory object on a particular execution path. AddressWatcher’s novelty lies in the fact that it is a dynamic approach that allows the semantics of a memory object to be tracked over multiple execution paths using a leak database. Our evaluation reveals that AddressWatcher correctly suggests fixes for 23 out of 50 memory leaks in five benchmark open-source projects. Finally, we demonstrate AddressWatcher’s practicality and relevance to the open-source community by submitting 25 PRs to 12 popular open-source repositories to fix memory leaks. Out of these submissions, 21 PRs have been successfully approved and incorporated within the respective codebases. Furthermore, one of our fixes was deemed so critical that it prompted a new version release for the calc repository.

## REFERENCES

- [1] “Denial Of Service,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: [https://owasp.org/www-community/vulnerabilities/Memory\\_leak](https://owasp.org/www-community/vulnerabilities/Memory_leak)
- [2] “CVE - CVE-2022-27819,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-27819>
- [3] “CVE - CVE-2017-1081,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1081>
- [4] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, “Linux kernel vulnerabilities: state-of-the-art defenses and open problems,” in *APSys ’11: Proceedings of the Second Asia-Pacific Workshop on Systems*. New York, NY, USA: Association for Computing Machinery, Jul. 2011, pp. 1–5.
- [5] “CWE - CWE-400: Uncontrolled Resource Consumption (4.11),” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://cwe.mitre.org/data/definitions/400.html>
- [6] “Memory leak | OWASP Foundation,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: [https://owasp.org/www-community/vulnerabilities/Memory\\_leak](https://owasp.org/www-community/vulnerabilities/Memory_leak)
- [7] J. Lee, S. Hong, and H. Oh, “MemFix: static analysis-based repair of memory deallocation errors for C,” in *ESEC/FSE 2018: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 95–106.
- [8] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, “Safe memory-leak fixing for C programs,” in *ICSE ’15: Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, May 2015, pp. 459–470.
- [9] H. Yan, Y. Sui, S. Chen, and J. Xue, “AutoFix: an automated approach to memory leak fixing on value-flow slices for C programs,” *SIGAPP Appl. Comput. Rev.*, vol. 16, no. 4, pp. 38–50, Jan. 2017.
- [10] B. Yu, C. Tian, N. Zhang, Z. Duan, and H. Du, “A dynamic approach to detecting, eliminating and fixing memory leaks,” *J. Comb. Optim.*, 2021. [Online]. Available: <https://www.semanticscholar.org/paper/A-dynamic-approach-to-detecting%2C-eliminating-and-Yu-Tian/c6348fbfd0805a3b539be9498896a53e47980d65>
- [11] J. Clause and A. Orso, “LEAKPOINT: pinpointing the causes of memory leaks,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*. IEEE, May 2010, vol. 1, pp. 515–524.
- [12] Contributors to Wikimedia projects, “SAT solver - Wikipedia,” Apr. 2023, [Online; accessed 4. May 2023]. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=SAT\\_solver&oldid=1150878317](https://en.wikipedia.org/w/index.php?title=SAT_solver&oldid=1150878317)
- [13] G. Xu, M. D. Bond, F. Qin, and A. Rountev, “LeakChaser: helping programmers narrow down causes of memory leaks,” in *ACM SIGPLAN Notices*. New York, NY, USA: Association for Computing Machinery, Jun. 2011, vol. 46, no. 6, pp. 270–282.
- [14] “ERROR: LeakSanitizer: detected memory leaks · Issue #123 · ndevilla/iniparser,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/ndevilla/iniparser/issues/123>
- [15] “Memory leak in function ‘h2o\_socket\_log\_tcp\_congestion\_controller’ · Issue #3354 · h2o/h2o,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/h2o/h2o/issues/3354>
- [16] “tests: Memory leaks in testing/fulltests/default/T115agentxperl\_simple · Issue #723 · net-snmp/net-snmp,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/net-snmp/net-snmp/issues/723>
- [17] “Harmless memory leak in split\_server\_port() · Issue #165 · rfc1036/whois,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/rfc1036/whois/issues/165>

- [//github.com/rfc1036/whois/issues/165](https://github.com/rfc1036/whois/issues/165)
- [18] “MemLeak in parse conf · Issue #1652 · nanomq/nanomq,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/nanomq/nanomq/issues/1652>
- [19] “AddressWatcher,” Sep. 2023, [Online; accessed 18. Sep. 2023]. [Online]. Available: <https://github.com/ashamedbit/AddressWatcher>
- [20] D. Bruening and Q. Zhao, “Practical memory checking with Dr. Memory,” in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, Apr. 2011, pp. 213–223.
- [21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: a fast address sanity checker,” in *USENIX ATC’12: Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association, Jun. 2012, p. 28.
- [22] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.
- [23] E. Stepanov and K. Serebryany, “MemorySanitizer: Fast detector of uninitialized memory use in C++,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Feb. 2015, pp. 46–55.
- [24] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, “Dynamic Race Detection with LLVM Compiler,” in *Runtime Verification*. Berlin, Germany: Springer, 2012, pp. 110–114.
- [25] A. Saeed, A. Ahmadinia, and M. Just, “Tag-protector: an effective and dynamic detection of illegal memory accesses through compile time code instrumentation,” *Advances in Software Engineering*, vol. 2016, 2016.
- [26] google, “sanitizers,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer>
- [27] Y. Sui, D. Ye, and J. Xue, “Static memory leak detection using full-sparse value-flow analysis,” in *ISSTA 2012: Proceedings of the 2012 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, Jul. 2012, pp. 254–264.
- [28] openssh, “openssh-portable,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://github.com/openssh/openssh-portable>
- [29] bminor, “binutils-gdb,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://github.com/bminor/binutils-gdb>
- [30] tmux, “tmux,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://github.com/tmux/tmux>
- [31] openssl, “openssl,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://github.com/openssl/openssl>
- [32] git, “git,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <https://github.com/git/git>
- [33] bminor, “binutils-gdb,” May 2023, [Address-Watcher failure due to Error Paths]. [Online]. Available: <https://github.com/bminor/binutils-gdb/commit/a26a013f22a19e2c16729e64f40ef8a7dfcc086e>
- [34] —, “binutils-gdb,” May 2023, [AddressWatcher failure due to Code organization]. [Online]. Available: <https://github.com/bminor/binutils-gdb/commit/7ed1aca0b5d135342f9dcc0eb0387dff95005a>
- [35] “MemFix Project,” May 2023, [Online; accessed 4. May 2023]. [Online]. Available: <http://prl.korea.ac.kr/MemFix>
- [36] “calc-2.15.0.6 release,” Feb. 2024, [Online; accessed 28. Feb. 2024]. [Online]. Available: <https://github.com/lcn2/calc/discussions/143>
- [37] “Proposal / discussion: Build all of coturn as C++17, instead of C11. by jonesmz · Pull Request #1416 · coturn/coturn,” Feb. 2024, [Online; accessed 28. Feb. 2024]. [Online]. Available: <https://github.com/coturn/coturn/pull/1416>
- [38] “Fix memory leak in whois.c by ashamedbit · Pull Request #168 · rfc1036/whois,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/rfc1036/whois/pull/168>
- [39] “Fix leak in socket.c by ashamedbit · Pull Request #3359 · h2o/h2o,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/h2o/h2o/pull/3359>
- [40] “Fix memory leak in mx.c by ashamedbit · Pull Request #4185 · neomutt/neomutt,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/neomutt/neomutt/pull/4185>
- [41] “calc,” Feb. 2024, [Online; accessed 28. Feb. 2024]. [Online]. Available: <https://github.com/lcn2/calc>
- [42] “Fix memory leak in zrandom.c by ashamedbit · Pull Request #142 · lcn2/calc,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/lcn2/calc/pull/142>
- [43] “calc/CHANGES at master · lcn2/calc,” Mar. 2024, [Online; accessed 4. Mar. 2024]. [Online]. Available: <https://github.com/lcn2/calc/blob/master/CHANGES>
- [44] “radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2>
- [45] “Fix leak due to r\_bin\_field\_free by ashamedbit · Pull Request #22643 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22643>
- [46] “Fix leak in esil.c by ashamedbit · Pull Request #22642 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22642>
- [47] “Fix leak in swift-sd by ashamedbit · Pull Request #22641 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22641>
- [48] “Fix leaks in bin\_elf.inc.c by ashamedbit · Pull Request #22638 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22638>
- [49] “Fix leaks related to name of RBinImport by ashamedbit · Pull Request #22629 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22629>
- [50] “Fix leak in cmd\_meta.inc.c by ashamedbit · Pull



- Request #22627 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22627>
- [51] “Fix leaks in profile.c by ashamedbit · Pull Request #22622 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22622>
- [52] “Fix leak in lib.c by ashamedbit · Pull Request #22621 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22621>
- [53] “Free leak in reg.c by ashamedbit · Pull Request #22620 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22620>
- [54] “Fix leak in canvas.c by ashamedbit · Pull Request #22655 · radareorg/radare2,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/radareorg/radare2/pull/22655>
- [55] “coturn,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/coturn/coturn>
- [56] “Fix memory leak on http\_server.c by ashamedbit · Pull Request #1412 · coturn/coturn,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/coturn/coturn/pull/1412>
- [57] “Fix memory leak in netengine.c by ashamedbit · Pull Request #1411 · coturn/coturn,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/coturn/coturn/pull/1411>
- [58] “Fix memory leak in rfc5769check.c by ashamedbit · Pull Request #1410 · coturn/coturn,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/coturn/coturn/pull/1410>
- [59] “net-snmp,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/net-snmp/net-snmp?tab=readme-ov-file>
- [60] “Fix leak in snmpusm.c by ashamedbit · Pull Request #792 · net-snmp/net-snmp,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/net-snmp/net-snmp/pull/792>
- [61] “Fix leaks in snmpv3.c by ashamedbit · Pull Request #791 · net-snmp/net-snmp,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/net-snmp/net-snmp/pull/791>
- [62] “neomutt,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/neomutt/neomutt>
- [63] “whois,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/rfc1036/whois>
- [64] “h2o,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/h2o/h2o>
- [65] “iniparser,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/ndevilla/iniparser>
- [66] “Free dictionaries in test\_dictionary.c by ashamedbit · Pull Request #151 · ndevilla/iniparser,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/ndevilla/iniparser/pull/151>
- [67] “Correct a memory leak problem by zrrto · Pull Request #114 · ndevilla/iniparser,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/ndevilla/iniparser/pull/114>
- [68] “NanoNNG,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/nanomq/NanoNNG>
- [69] “Fix memory leak in hocon.c by ashamedbit · Pull Request #853 · nanomq/NanoNNG,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/nanomq/NanoNNG/pull/853>
- [70] “Fix memory leak in nasm-parse.c by ashamedbit · Pull Request #265 · yasm/yasm,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/yasm/yasm/pull/265>
- [71] “Fix memory leak in nasm-pp.c by ashamedbit · Pull Request #264 · yasm/yasm,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/yasm/yasm/pull/264>
- [72] “Fix memory leaks in write\_c and eval\_shell by ashamedbit · Pull Request #165 · neurobin/shc,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/neurobin/shc/pull/165>
- [73] “Fix leak in makemon.c by ashamedbit · Pull Request #533 · elunna/hackem,” Mar. 2024, [Online; accessed 6. Mar. 2024]. [Online]. Available: <https://github.com/elunna/hackem/pull/533>
- [74] S. Cherem, L. Princehouse, and R. Rugina, “Practical memory leak detection using guarded value-flow analysis,” *SIGPLAN Not.*, vol. 42, no. 6, pp. 480–491, Jun. 2007.
- [75] Y. Jung and K. Yi, “Practical memory leak detector based on parameterized procedural summaries,” in *ISMM ’08: Proceedings of the 7th international symposium on Memory management*. New York, NY, USA: Association for Computing Machinery, Jun. 2008, pp. 131–140.
- [76] W. Li, H. Cai, Y. Sui, and D. Manz, “PCA: memory leak detection using partial call-path analysis,” in *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1621–1625.
- [77] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, “MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks,” *arXiv*, Mar. 2022.
- [78] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, “SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, May 2019, pp. 72–82.
- [79] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.
- [80] C. Lattner, A. Lenharth, and V. Adve, “Making context-sensitive points-to analysis with heap cloning practical for the real world,” *SIGPLAN Not.*, vol. 42, no. 6, pp. 278–289, Jun. 2007.



**Aniruddhan Murali** is a Ph.D. candidate in the Cheriton School of Computer Science at the University of Waterloo, Canada. His research interests include fuzzing, code slicing, vulnerability detection, and automatic bug fixing. You can find more about him [here](#).



**Mahmoud Alfadel** is a postdoctoral researcher in the Cheriton School of Computer Science, University of Waterloo. His research interests include mining software repositories, empirical software engineering, software ecosystems, open-source security, and release engineering. You can find more about him at <https://rebels.cs.uwaterloo.ca/member/mahmoud.html>



**Meiyappan Nagappan** is an Associate Professor at the Cheriton School of Computer Science, University of Waterloo. He has worked on empirical software engineering to address software development concerns and currently researches the impact of large language models on software development.



**Meng Xu** is an Assistant Professor in the Cheriton School of Computer Science at the University of Waterloo, Canada. His research is in the area of system and software security, with a focus on delivering high-quality solutions to practical security programs, especially in finding and patching vulnerabilities in critical computer systems. This usually includes research and development of automated program analysis / testing / verification tools that facilitate the security reasoning of critical programs.



**Chengnian Sun** is currently an Associate Professor at the Cheriton School of Computer Science, University of Waterloo, Canada. His research interests include software engineering and programming languages, with a focus on techniques, tools, and methodologies for improving software quality and developer productivity. He received his Ph.D. degree from the School of Computing at the National University of Singapore.