AddressWatcher: Sanitizer-Based Localization of Memory Leak Fixes

Aniruddhan Murali, *Student Member, IEEE*, Mahmoud Alfadel, *Member, IEEE*, Meiyappan Nagappan, Meng Xu, *Member, IEEE*, and Chengnian Sun, *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 paths (path explosion). On the other hand, dynamic approaches can spell out precise semantics of memory object only on a single execution path (it does not consider 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. Addresswatcher accomplishes this by using a leak database that allows one to store and compare different execution paths of a leak over several test cases. Also, AddressWatcher performs lightweight instrumentation during compile time that is utilized during the program execution to watch and track memory leak read/writes. We conduct an evaluation of AddressWatcher over five popular packages, namely binutils, openssh, tmux, openssl and git. In 23 out of 50 real-world memory leak bugs, AddressWatcher correctly points to a free location to fix memory leaks. Finally, we submit 25 Pull Requests across 12 popular OSS repositories using AddressWatcher suggestions. Among these, 21 were merged leading to 5 open issues being addressed. In fact, our 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.**

***Index Terms*—Memory leak, dynamic analysis, vulnerability.**

I. 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 [6]. Recently, a number of memory leak vulnerabilities have been disclosed in Linux kernel (e.g., CVE-2022-27819 [4], CVE-2017-10810 [3]). Such vulnerabilities in the kernel had severe consequences on system stability and availability [5], [8], [58].

Fixing memory leaks manually is often time-consuming and error-prone for developers [69]. Hence, prior work focused on designing and implementing techniques to address challenges of automatically fixing memory leak bugs [60], [64], [69], [81], [82]. Much of these prominent techniques leverage static and dynamic approaches to fixing memory leak bugs, although very few of these approaches have been open-sourced [64], [69] (both of them being static approaches).

An exemplary static analysis tool is Memfix [69]. Memfix identifies all paths involving an allocated memory. It models the problem of identifying a set of deallocation statements as an exact cover problem. It then uses a SAT solver to find the solution to the exact cover problem [62]. The solution suggested by Memfix is always a safe fix. However, Memfix cannot identify program paths in the presence of function pointers and recursion and 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 [60]. 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

Manuscript received 20 October 2023; revised 9 July 2024; accepted 17 July 2024. Date of publication 5 August 2024; date of current version 19 September 2024. Recommended for acceptance by L. Mariani. *(Corresponding author: Aniruddhan Murali.)*

The authors are with David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3K4, Canada (e-mail: a25murali@uwaterloo.ca; malfadel@uwaterloo.ca; mei.nagappan@uwaterloo.ca; meng.xu.cs@uwaterloo.ca; cnsun@uwaterloo.ca).

Digital Object Identifier 10.1109/TSE.2024.3438119

0098-5589 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

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:

- 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 [69], Leakfix [64]).
- 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 [60], LeakChaser [80]).

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 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 utilizing sanitizer-based light-weight compile time instrumentation.

Contributions. The contributions of this work 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 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 [69].
3) We showcase the practicality and relevance of AddressWatcher by submitting 25 Pull Requests (PRs) to prominent open-source projects that are affected by memory leak bugs. Out of these submissions, 21 PRs have been successfully approved and merged (i.e., the fixes provided by AddressWatcher have been applied), with an additional 4 PRs awaiting review. Furthermore, our accepted PRs have resulted in the resolution of 5 open issues within these projects [14], [41], [43], [44], [50]. Additionally, the critical nature of our fix led to a new version release for the calc repository [1] and intense discussions on memory safety in the coturn repository [48].
4) We develop a prototype tool for AddressWatcher. The tool and benchmarks are made publicly available [2].

II. Memory Leak & 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 [56], [71], [75], [77]. **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 [74], [76]. **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 as shown in Fig. 1. 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 [75], Dr. Memory [56]) have been proposed to detect such scenarios by utilizing 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 [66] 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.

Leak detection vs fixing. While memory leak detection and leak fixing are relevant, they require fundamentally different

```
1   void read(int size)
2   {
3       char* p = (char*)malloc(size);
4       fgets(p, size, stdin);
5       if (*p == "\n")
6       {
7          // Path 1: Abrupt return
8          // Leak p
9          return;
10      }
11      // Path 2: Process p
12      // Leak p
13      return;
14  }
```

Listing 1: Leak detection vs leak fixing.

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.



Fig. 1. Redzones guarding heap allocations. These redzones contain allocation stack traces and allocator thread ID.

approaches to address them. A dynamic leak detection technique such as LSAN [66] checks for memory that is allocated but not freed only at program termination. It does not profile leaked memory because it is an additional overhead and is not required to detect a leak. In the case of static leak detectors such as Saber [78], 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).

III. Design

AddressWatcher is an approach that performs memory tracking to automatically suggest a location 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 $o_1, o_2, o_3....o_n$ 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 design components:

(D1) **Instrument relevant code points.** Set up checkpoints at all relevant code points through binary instrumentation iie. $o_1, o_2, ...o_n$.

(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 IV, we discuss how each of these challenges is handled.

IV. 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.

Fig. 2 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 outlines 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 Fig. 2. 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 Fig. 2, 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 read 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 logics 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

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

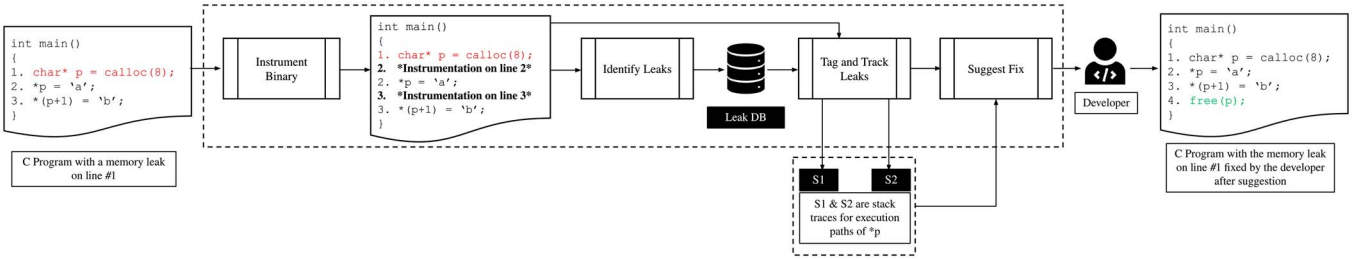


Fig. 2. An overview of our approach for suggesting a location of a memory leak fix.

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 II). For the running example in Fig. 2, 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 test-case which accomplishes design step (D2). In fact, the checker additionally stores the allocation stack traces of the leaks into a database which we call 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 design component (D5). Subsequent testcases will then read from leak database to match allocations with leaks from testcases in the past. Such a database aims to help us for profiling and tracking 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 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 Fig. 2, 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 Fig. 2 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 Fig. 2. Note that we provide more details about the special value of shadow memory and tracking process in Technical Challenges Section V.

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 Fig. 2, 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 iie. we find the last use point over all execution traces stored in leak database. Note that if

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

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 frees are required for a given memory leak. A detailed explanation of the AddressWatcher algorithm that handles multiple deallocations is described in Technical Challenges Section V. This step enables the realization of the design component (D6).

Automated fix insertion. We additionally provide a script patch.py [9] which performs a deallocation using 'free' function. The script takes the Addresswatcher leaked execution profile as input and generates the free statement. The goal of the script is to identify the variable to be freed after which it inserts 'free(leaked_variable)' at the AddressWatcher fix location. Referring to the code in Fig. 2, in order to achieve this, the script performs the following steps:

1) Identifies all statements that are a part of the execution profile. The leaked execution profile consists of lines 1, 2 and 3.
2) Identifies variables in the statements in the previous step through the AST. We use srcml [9] for extracting the AST. In our running example, p is only variable on all lines 1, 2 and 3.
3) Traces the leaked variable to last use point through common variable names between successive statements in execution profile. In our example, the leaked variable from the allocation line is p.
4) Finally, the script inserts 'free(leaked_variable)' at the AddressWatcher fix location. The constructed fix is 'free(p)' in our example.

A developer may chose to manually construct the fix without using the script in order to patch the leak using a custom API at the same fix location suggested by AddressWatcher. Therefore, we provide developers with this feature as an optional utility of AddressWatcher. An example video of fixing the leak using this feature is shown here [52].

V. 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 [75] and LSAN [66] runtime libraries within `gcc` compiler.

Detecting leaks. We use a sanitizer for detecting memory leaks - LSAN [66]. During compile time, we compile with LSAN option, which add a leak checker module to identify leaks in the program.

Constant memory address offset. Throughout our approach, we re-execute a given program with multiple testcases. The goal of the first run is to identify memory leaks in the program (step 2, Section IV). Subsequent runs aim to track the already identified leaks (step 3, Section IV). In fact, modern systems adopt a technique called ASLR (Address Space Layout Randomisation) [61] which randomizes memory addresses on different binary runs to increase the difficulty of attackers for exploiting buffer overflows. Hence, different runs of a binary program would have different (random) memory address offsets. In our approach, it is essential to maintain one constant memory address offset throughout all runs, and therefore, we disable ASLR to maintain consistency of stack traces.

Implementing leaks database. A key component in our approach is the leak database (step 2, Section IV). 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. This ensures that sensitive program data in the leak database is not accessible to other users.

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 Fig. 2, AddressWatcher suggests adding a fix after line 3. Let us assume that a developer agrees with the suggestion and inserts a free statement at 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 it's 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 [56], [71], [75], [77] that utilize shadow memory to detect memory errors, AddressWatcher utilizes shadow memory to tag leaks on different executions and to finally suggest leak fixes. This is a unique design aspect of AddressWatcher.

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

Listing 2: ASAN instrumentation.

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

TABLE I
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

Implementing leak tracking. AddressWatcher utilizes instrumentation to perform leak tracking. This instrumentation is a modification of ASAN instrumentation [75].

Before reads and writes 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 utilize the function `Report` to track execution paths of leaked objects (step 3, Section IV). 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.

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 I. 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 (iie. 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 it's last code point (iie. 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 it's fixes by storing relevant execution traces in the leak database for analysis. When the test suite covers *all* the leaked paths, AddressWatcher will be able to suggest the correct patch for the memory leak.

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. We leave deallocation using custom APIs as a direction for future research.

Combining ASAN and LSAN. AddressWatcher modifies and utilizes components of ASAN and LSAN in order to suggest memory leak fixes. ASAN is originally designed to detect buffer overflows through instrumentation and shadow memory. LSAN only detects memory leaks. These tools individually do not suggest or provide hints for memory leak fixing. AddressWatcher achieves this by using LSAN to detect a memory leak first and automatically stores it in a leak database. Subsequently, AddressWatcher utilizes modified instrumentation of ASAN to also track the stored leak from the leaks database. AddressWatcher cannot accomplish it's goal with only either of these tools.

VI. EVALUATION

In this section, we evaluate our proposed approach. In particular, we introduce our evaluation dataset in Section VI-A and present how fixes suggested by AddressWatcher compare with manual fixes from developers in Section VI-B. Next, we conduct a comparison with Memfix tool, the state of the art approach for statically fixing memory leaks, in Section VI-C. We then present the efficiency evaluation of AddressWatcher in Section VI-D. 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 VI-E.

A. 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 [69]. The benchmark used by Memfix includes memory leak bugs in five open source packages, namely, `openssh`, `binutils`, `tmux`, `openssl` and `git` [53], [65], [72], [73], [79].

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

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

TABLE II
LIST OF 50 BUG FIXING COMMITS IN OUR BENCHMARK, PER PACKAGE. FOR EACH COMMIT, WE PRESENT: 1) WHETHER ADDRESSWATCHER IS ABLE TO FIX THE BUG, 2) THE REASON ADDRESSWATCHER FAILS TO FIX THE BUG, 3) WHETHER MEMFIX IS ABLE TO FIX THE BUG

Repository	Leak No.	Fixed by AW (✓/×)	AW Failure Reason	Fixed by MF? (✓/×)
binutils	1	✓	-	✓
	2	✓	-	✓
	3	✓	-	✓
	6	✓	-	✓
	4	✓	-	×
	5	✓	-	×
	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	×

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.

B. AddressWatcher Effectiveness

Table II presents the distribution of 50 bug fixing commits across the three studied packages in our dataset. For each fixing commit, Table II 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 II 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` [54]. 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 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 [54].

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`

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

```
1   struct btrace
2   {
3       char* p;
4       char* q;
5   }
6   void btrace_alloc (struct* X)
7   {
8       X = malloc(sizeof(X));
9       X->p = malloc(10);
10      X->q = malloc(10);
11  }
12  void btrace_clear (struct* X)
13  {
14      // Developer adds free(X->p) here
15      free(X->q)
16      free(X)
17  }
18  int main ()
19  {
20      struct btrace* X;
21      btrace_alloc(X); // Allocate X,p,q
22      doSomething1(X->p); // Use X->p
23      // AddressWatcher suggests fix for X->p here
24      doSomething2(X->q); // Use X->q
25      btrace_clear(X); // Deallocate X,q
26      return 0;
27  }
```

Listing 4: Code snippet from binutils showing AddressWatcher failure due to code organization [55].

[55], which is illustrated in Listing 4. As the Listing shows, there is a memory leak originating from 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 6) which is used for allocation, and the other is called `btrace_clear` (on line 12) 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 14 in `btrace_clear` before the struct X 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.

C. 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 [69]. 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, and how AddressWatcher can be improved in the light of our comparison.

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 [62]. Then, all frees that are present in these paths are removed, and new free statements are added.

TABLE III
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

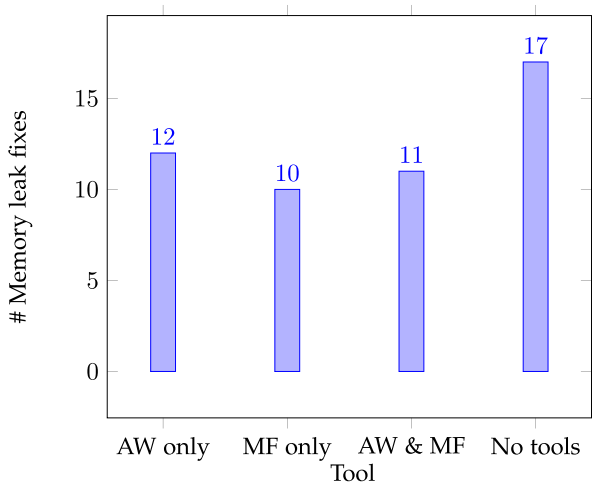


Fig. 3. Distribution of memory leak fixes by AddressWatcher (AW) and Memfix (MF).

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 [7]. We run Memfix without any time constraints and compare results with AddressWatcher.

Table III 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 Fig. 3. 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 that classify

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

these bugs within the following four scenarios (S1 – S4) under which each approach (static vs. dynamic) succeeds or fails to fix memory leaks:

S1. Memory leak bugs fixed by both AddressWatcher and Memfix. For the 11 cases solved by both AddressWatcher and Memfix, we find that the allocated and leaked variable is largely limited in scope within a procedure and has limited leaked paths. All of these leaked paths are also easily covered by simple test cases. Hence, both approaches are able to fix them.

S2. 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.

S3. 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 VI-B.

S4. 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 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.

D. 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 IV.

TABLE IV
EFFICIENCY OF ADDRESSWATCHER (AW) AND MEMFIX (MF)

Repository	% AW Overhead	AW Avg Run time (s)	AW Max Run Time (s)	MF Avg Run Time (s)	MF Max Run Time (s)
binutils	178.24	0.53	0.62	0.40	1.11
tmux	160.69	0.51	0.54	1.02	5.00
openssh-portable	142.75	0.52	0.75	1.58	2.23
openssl	147.32	0.50	0.58	11.51	113.18
git	171.36	0.48	0.60	9.50	87.52

Table IV 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. Similarly, we show the average runtime for Memfix per repository and the maximum runtime for Memfix per leak in a given repository. From the table, we observe that Memfix quickly generates fixes or errors out in certain repositories such as binutils, tmux and openssh. However, openssl and git contain complex interprocedural leaks, which require significant time by their SAT solver, often leading to an error. We emphasize that Memfix is a static tool that analyzes code to suggest leak fixes, while AddressWatcher instead executes testcases to suggest leak fixes. We refrain from using Memfix as a baseline for efficiency comparison with AddressWatcher. We report on this analysis only for transparency purposes, and we do not induce any specific insights.

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.

E. 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 using the GitHub search functionality. Subsequently, we 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 issues on memory leaks being resolved [14], [41], [43], [44], [50]. One

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

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 [1]. 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 [48]. 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:

```
    sockfd = openconn(server, port);
    free(server); // AddressWatcher fix
    server = do_query(sockfd, query_string);
```

Listing 5: whois PR showing AddressWatcher leak fix [34].

```
  if (getsockopt(fd, IPPROTO_TCP, TCP_CONGESTION,
    buf, &buflen) == 0) {
      return h2o_iovec_init(buf, strlen(buf));
  }
  free(buf); // AddressWatcher fix
```

Listing 6: h2o PR showing AddressWatcher leak fix [22].

```
    // Displayed if appending to trash fails when
    syncing or closing a mailbox
    if (mutt_append_message(m_trash, m, e, NULL,
    MUTT_CM_NO_FLAGS, CH_NO_FLAGS) == -1)
    {
        mx_mbox_close(m_trash);
        mailbox_free(&m_trash); // AW fix
        return -1;
    }
```

Listing 7: neomutt PR showing AddressWatcher fix [29].

1) Calc (1 PR merged): Calc [10] 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. The repository owners mentioned they have actively been trying to patch a memory leak in calc because it could be quite deadly over such long periods of execution time. We merged a PR in this repository [35] and were given credit for fixing a "long-standing memory leak" on their discussion forum [1] and CHANGES document [11]. 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 [1].
2) Radare2 (10 PRs merged): It is a library used to ease binary reverse engineering tasks and binary exploitation [49]. We submitted 10 PRs fixing leaks detected while analyzing "ls" binary and by running the visual mode of radare [15], [16], [17], [18], [19], [23], [24], [25], [27], [39]. Three of these PRs fixed leaks across multiple locations as well [23], [24], [27]. All 10 were merged. In one PR, the author was even impressed with a fix saying "whoa nice spot!" [16].
3) CoTurn (3 PRs merged): Coturn is a free open source implementation of a VoIP media traffic NAT traversal server and gateway [13]. We submitted 3 PRs to this repository and all 3 were merged [32], [33], [36]. The PRs that we submitted initiated a long discussion on how to migrate project code to C++17 to improve memory safety using destructors [48]. Later after submitting the PRs, the first author was approached by mail for a consultancy position for optimizing coturn as well.
4) Net-snmp (2 PRs merged): This library is a widely used protocol for monitoring the health and welfare of network equipment [47]. We fixed 2 leaks identified by fuzzers when the '-a' argument of net-snmp is used incorrectly [21], [26]. Both PRs were merged. One open issue was resolved [50].
5) Neomutt (1 PR merged): NeoMutt is a command line mail reader [46]. We submitted a PR fixing a memory leak when certain mail is added to trash [29], as shown in Listing 7. In this case, the trash mailbox is closed, but the memory associated with it is not freed. AddressWatcher 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 command-line utility on most UNIX systems used to make WHOIS protocol queries [51]. 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 [34] and resolved another issue [41].
7) h2o (1 PR merged): h2o is a really fast HTTP server [40]. 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 [22] and resolved another issue [44].
8) Iniparser (1 PR merged): This library offers parsing of ini files from C [42]. We submitted a PR [38] which project owners mentioned to be more comprehensive than another PR submitted by a different user [12]. Our PR was merged in the repository leading to an open issue being resolved [14].
9) NanoNNG (1 PR merged): NanoNNG is a lightweight library, offering a simple API to solve common recurring messaging problems, such as publish/subscribe or RPC-style request/reply [45]. It is used in multiple IoT devices where memory management is crucial. We merged a PR that fixed a memory leak that occurred on reading configuration files [28]. This also led to an open issue being closed [43].

We also submitted 4 PRs in 3 other repositories. We submitted 2 PRs in yasm repository [30], [31], 1 PR in shc repository [37], and 1 PR in hackem repository [20]. However, these PRs are still under review and are yet to be merged. Overall, the strength of AddressWatcher lies in its automatic capability to localize leak-fix locations across 25 PRs.

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

VII. Discussion

In the previous Section VI, we discuss cases where AddressWatcher fails to fix memory leaks. This section discusses some aspects that impact the performance of AddressWatcher when suggesting fix locations.

Code coverage. AddressWatcher fundamentally relies on 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 [60], [80], [82]. A 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 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.

VIII. 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.

A. Approaches for Memory Leak Detection

Static approaches. A plethora of recent work focused on detecting memory leaks statically [57], [59], [67], [70], [78]. For example, Cherem et al. and Sui et al. propose Fastcheck [59] and Saber [78], respectively. These techniques perform pointer value-flow analysis from allocations to deallocation sites. More specifically, they use a sparse graph representation to capture pointer define-use chains. Memory leaks detection is hence reduced to solving graph reachability queries on the constructed graph.

Static analysis approaches such as Smoke [63] 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 which performs dynamic analysis for detecting memory leaks [66]. When memory is freed, a magic value is written into the memory. When program exits, the heap is checked for memory leaks, i.e., by identifying allocated memory that have not been overwritten with a magic value. Nethercote et al. in their framework Valgrind [71] 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.

B. Approaches for Memory Leak Fixing

Static approaches. Some prominent techniques have been proposed to statically suggest fixes for memory leaks [64], [69]. For example, Memfix by Lee et al. [69] 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 modelled as 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 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 [64]. 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 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

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

[68].) 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 [60], [80], [82]. Yu et al. in their tool DEF_LEAK propose a dynamic symbolic execution approach to expose memory leaks occurring in all execution paths [82]. 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 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 test suite. Autofix [81] is a static and dynamic analysis tool that uses static value flow analysis from allocated memory to identify potential locations where memory can be freed. It then instruments code in these locations where memory is automatically freed. The instrumentation in Autofix is added based on static value flow analysis in Saber [78], which is neither sound nor complete- bounding loops and recursion to utmost one iteration and does not capture path correlations. AddressWatcher adds instrumentation wherever heap variables are read/written and does not suffer from an imprecise value flow analysis.

The work most close to ours is proposed by Clause et al. where they proposed LeakPoint [60]. 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 [60]. We showed in Section VI-D 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 these dynamic tools, which makes it infeasible to directly compare to AddressWatcher. AddressWatcher on the other hand is open source with a publicly available docker [2].

IX. 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 object 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 23 out of 50 real-world bugs in five diverse open-source projects. Our further analysis indicates that AddressWatcher can be considered complementary to existing approaches to memory leak fixing. Finally, we demonstrate AddressWatcher's practicality and relevance to the open-source community by submitting 25 PRs to 12 popular open-source repositories. Out of these submissions, 21 PRs have been successfully approved and merged (i.e., the fixes provided by AddressWatcher have been applied and incorporated with the codebase). Furthermore, our accepted PRs have resulted in the resolution of 5 open issues within these projects.

References

[1] "calc-2.15.0.6 release." GitHub. Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/lcn2/calc/discussions/143

[2] "AddressWatcher." GitHub. Accessed: Sep. 18, 2023. [Online]. Available: https://github.com/ashamedbit/AddressWatcher

[3] "CVE - CVE-2017–1081." MITRE Corporation. Accessed: May 4, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1081

[4] "CVE - CVE-2022–27819." MITRE Corporation. Accessed: May 4, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-27819

[5] "CWE - CWE-400: Uncontrolled resource consumption (4.11)." MITRE Corporation. Accessed: May 4, 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/400.html

[6] "Denial of service." OWASP. Accessed: May 4, 2023. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Memory_leak

[7] "MemFix project." Accessed: May 4, 2023. [Online]. Available: http://prl.korea.ac.kr/MemFix

[8] "Memory leak | OWASP Foundation." OWASP. Accessed: May 4, 2023. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Memory_leak

[9] "AddressWatcher/benchmarks/patch.py at main · ashamedbit/AddressWatcher." GitHub. Accessed: Jul. 6, 2024. [Online]. Available: https://github.com/ashamedbit/AddressWatcher/blob/main/benchmarks/patch.py

[10] "calc." GitHub. Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/lcn2/calc

[11] "calc/CHANGES at master · lcn2/calc." GitHub. Accessed: Mar. 4, 2024. [Online]. Available: https://github.com/lcn2/calc/blob/master/CHANGES

[12] "Correct a memory leak problem by zrrto · Pull Request #114 · ndevilla/iniparser." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/ndevilla/iniparser/pull/114

[13] "coturn." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/coturn/coturn

[14] "ERROR: LeakSanitizer: Detected memory leaks · Issue #123 · ndevilla/iniparser." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/ndevilla/iniparser/issues/123

[15] "Fix leak due to r_bin_field_free by ashamedbit · Pull Request #22643 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22643

[16] "Fix leak in canvas.c by ashamedbit · Pull Request #22655 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22655

[17] "Fix leak in cmd_meta.inc.c by ashamedbit · Pull Request #22627 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22627

[18] "Fix leak in esil.c by ashamedbit · Pull Request #22642 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22642

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

[19] "Fix leak in lib.c by ashamedbit · Pull Request #22621 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22621
[20] "Fix leak in makemon.c by ashamedbit · Pull Request #533 · elunna/hackem." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/elunna/hackem/pull/533
[21] "Fix leak in snmpusm.c by ashamedbit · Pull Request #792 · net-snmp/net-snmp." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/net-snmp/net-snmp/pull/792
[22] "Fix leak in socket.c by ashamedbit · Pull Request #3359 · h2o/h2o." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/h2o/h2o/pull/3359
[23] "Fix leak in swift-sd by ashamedbit · Pull Request #22641 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22641
[24] "Fix leaks in bin_elf.inc.c by ashamedbit · Pull Request #22638 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22638
[25] "Fix leaks in profile.c by ashamedbit · Pull Request #22622 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22622
[26] "Fix leaks in snmpv3.c by ashamedbit · Pull Request #791 · net-snmp/net-snmp." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/net-snmp/net-snmp/pull/791
[27] "Fix leaks related to name of RBinImport by ashamedbit · Pull Request #22629 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22629
[28] "Fix memory leak in hocon.c by ashamedbit · Pull Request #853 · nanomq/NanoNNG." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/nanomq/NanoNNG/pull/853
[29] "Fix memory leak in mx.c by ashamedbit · Pull Request #4185 · neomutt/neomutt." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/neomutt/neomutt/pull/4185
[30] "Fix memory leak in nasm-parse.c by ashamedbit · Pull Request #265 · yasm/yasm." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/yasm/yasm/pull/265
[31] "Fix memory leak in nasm-pp.c by ashamedbit · Pull Request #264 · yasm/yasm." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/yasm/yasm/pull/264
[32] "Fix memory leak in netengine.c by ashamedbit · Pull Request #1411 · coturn/coturn." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/coturn/coturn/pull/1411
[33] "Fix memory leak in rfc5769check.c by ashamedbit · Pull Request #1410 · coturn/coturn." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/coturn/coturn/pull/1410
[34] "Fix memory leak in whois.c by ashamedbit · Pull Request #168 · rfc1036/whois." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/rfc1036/whois/pull/168
[35] "Fix memory leak in zrandom.c by ashamedbit · Pull Request #142 · lcn2/calc." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/lcn2/calc/pull/142
[36] "Fix memory leak on http_server.c by ashamedbit · Pull Request #1412 · coturn/coturn." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/coturn/coturn/pull/1412
[37] "Fix memory leaks in write_c and eval_shell by ashamedbit · Pull Request #165 · neurobin/shc." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/neurobin/shc/pull/165
[38] "Free dictionaries in test_dictionary.c by ashamedbit · Pull Request #151 · ndevilla/iniparser." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/ndevilla/iniparser/pull/151
[39] "Free leak in reg.c by ashamedbit · Pull Request #22620 · radareorg/radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2/pull/22620
[40] "h2o." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/h2o/h2o
[41] "Harmless memory leak in split_server_port() · Issue #165 · rfc1036/whois." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/rfc1036/whois/issues/165
[42] "iniparser." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/ndevilla/iniparser
[43] "MemLeak in parse conf · Issue #1652 · nanomq/nanomq." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/nanomq/nanomq/issues/1652
[44] "Memory leak in function 'h2o_socket_log_tcp_congestion_controller' · Issue #3354 · h2o/h2o." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/h2o/h2o/issues/3354
[45] "NanoNNG." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/nanomq/NanoNNG
[46] "neomutt." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/neomutt/neomutt
[47] "net-snmp." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/net-snmp/net-snmp?tab=readme-ov-file
[48] "Proposal / discussion: Build all of coturn as C++17, instead of C11. by jonesmz · Pull Request #1416 · coturn/coturn." GitHub. Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/coturn/coturn/pull/1416
[49] "radare2." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/radareorg/radare2
[50] "tests: Memory leaks in testing/fulltests/default/T115agentxperl_simple · Issue #723 · net-snmp/net-snmp." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/net-snmp/net-snmp/issues/723
[51] "whois." GitHub. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/rfc1036/whois
[52] M. Aniruddhan. "Patching memory leaks using AddressWatcher output." YouTube. Accessed: Jul. 6, 2024. [Online]. Available: https://www.youtube.com/watch?v=PIOPzy04eXw
[53] bminor. "binutils-gdb." GitHub. Accessed: May 4, 2023. [Online]. Available: https://github.com/bminor/binutils-gdb
[54] bminor. "binutils-gdb." GitHub. [AddressWatcher failure due to Error Paths]. Accessed: Jan. 21, 2024. [Online]. Available: https://github.com/bminor/binutils-gdb/commit/a26a013f22a19e2c16729e64f40ef8a7dfcc086e
[55] bminor. "binutils-gdb." GitHub. [AddressWatcher failure due to Code organization]. Accessed: Jan. 21, 2024. [Online]. Available: https://github.com/bminor/binutils-gdb/commit/7ed1acafa0b5d135342f9dcc0eb0387dff95005a
[56] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Piscataway, NJ, USA: IEEE Press, Apr. 2011, pp. 213–223.
[57] 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," 2022, *arXiv:2203.02660*.
[58] 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 *Proc. 2nd Asia-Pacific Workshop Syst. (APSys)*, New York, NY, USA: ACM, Jul. 2011, pp. 1–5.
[59] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 480–491, Jun. 2007.
[60] J. Clause and A. Orso, "LEAKPOINT: Pinpointing the causes of memory leaks," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, Piscataway, NJ, USA: IEEE Press, May 2010, vol. 1, pp. 515–524.
[61] Contributors to Wikimedia Projects. "Address space layout randomization - Wikipedia." Wikipedia. Accessed: May 4, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Address_space_layout_randomization&oldid=1150615839
[62] Contributors to Wikimedia pProjects. "SAT solver - Wikipedia." Wikipedia. Accessed: May 4, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SAT_solver&oldid=1150878317
[63] 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 *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, May 2019, pp. 72–82.
[64] Q. Gao et al., "Safe memory-leak fixing for C programs," in *Proc. 37th Int. Conf. Softw. Eng. (ICSE)*, vol. 1, Piscataway, NJ, USA: IEEE Press, May 2015, pp. 459–470.
[65] git. "git." GitHub. Accessed: May 4, 2023. [Online]. Available: https://github.com/git/git
[66] google. "sanitizers." GitHub. Accessed: May 4, 2023. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer
[67] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proc. 7th Int. Symp. Memory Manage. (ISMM)*, New York, NY, USA: ACM, Jun. 2008, pp. 131–140.
[68] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 278–289, Jun. 2007.
[69] J. Lee, S. Hong, and H. Oh, "MemFix: Static analysis-based repair of memory deallocation errors for C," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw.*Eng. *(ESEC/FSE)*, New York, NY, USA: ACM, Oct. 2018, pp. 95–106.

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.

[70] W. Li, H. Cai, Y. Sui, and D. Manz, "PCA: Memory leak detection using partial call-path analysis," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE),* New York, NY, USA: ACM, Nov. 2020, pp. 1621–1625.
[71] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, Jun. 2007.
[72] openssh. "openssh-portable." GitHub. Accessed: May 4, 2023. [Online]. Available: https://github.com/openssh/openssh-portable
[73] openssl. "openssl." GitHub. Accessed: May 4, 2023. [Online]. Available: https://github.com/openssl/openssl
[74] A. Saeed, A. Ahmadinia, and M. Just, "Tag-protector: An effective and dynamic detection of illegal memory accesses through compile time code instrumentation," *Adv. Softw. Eng.*, vol. 2016, no. 1, pp. 9842917–9842936, 2016.
[75] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX Association, Jun. 2012, p. 28.
[76] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler," in *Runtime Verification*. Berlin, Germany: Springer-Verlag, 2012, pp. 110–114.
[77] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in C++," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Piscataway, NJ, USA: IEEE Press, Feb. 2015, pp. 46–55.
[78] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, Jul. 2012, pp. 254–264.
[79] tmux. "tmux." GitHub. Accessed: May 4, 2023. [Online]. Available: https://github.com/tmux/tmux
[80] G. Xu, M. D. Bond, F. Qin, and A. Rountev, "LeakChaser: Helping programmers narrow down causes of memory leaks," in *Proc. ACM SIGPLAN Notices*, vol. 46, New York, NY, USA: ACM, Jun. 2011, no. 6, pp. 270–282.
[81] 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.
[82] B. Yu, C. Tian, N. Zhang, Z. Duan, and H. Du, "A dynamic approach to detecting, eliminating and fixing memory leaks," *J. Combinatorial Optim.*, vol. 42, pp. 409–426, 2021.



Aniruddhan Murali (Student Member, IEEE) is currently working toward the Ph.D. degree with David R. Cheriton School of Computer Science, University of Waterloo, Canada. His research interests include fuzzing, code slicing, vulnerability detection, and automatic bug fixing. For more information, see https://www.linkedin.com/in/aniruddhan-murali-40b41a11a/.



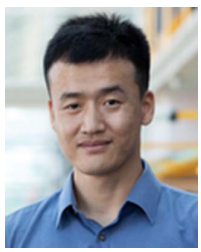
Mahmoud Alfadel (Member, IEEE) is a Postdoctoral Researcher with David R. Cheriton School of Computer Science, University of Waterloo. His research interests include mining software repositories, empirical software engineering, software ecosystems, open-source security, and release engineering. For more information, see https://rebels.cs.uwaterloo.ca/member/mahmoud.html.



Meiyappan Nagappan is an Associate Professor with David R. 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 (Member, IEEE) is an Assistant Professor with David R. Cheriton School of Computer Science, University of Waterloo, Canada. His research interests include 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 (Member, IEEE) received the Ph.D. degree from the School of Computing, National University of Singapore. He is an Associate Professor with David R. 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.

Authorized licensed use limited to: University of Calgary. Downloaded on October 30,2026 at 14:54:08 UTC from IEEE Xplore. Restrictions apply.