Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2013-IC-002

Dynamic Validating Static Memory Leak Warnings..

Mengchen Li Yuanjun Chen Linzhang Wang Guoqing Xu

Postprint Version. In Proceedings of
International Conference on Software Testing and Analysis (ISSTA 2013), Switzerland, ACM Press, 2013.

Dynamically Validating Static Memory Leak Warnings

Mengchen Li[1] Yuanjun Chen[1] Linzhang Wang[1] Guoqing Xu[2]
[1]State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, 210023, China
{limengchen, chenyj}@seg.nju.edu.cn lzwang@nju.edu.cn

[2]Department of Computer Science
University of California, Irvine, CA 92697, USA
guoqingx@ics.uci.edu

ABSTRACT

Memory leaks have significant impact on software availability, performance, and security. Static analysis has been widely used to find memory leaks in C/C++ programs. Although a static analysis is able to find all potential leaks in a program, it often reports a great number of false warnings. Manually validating these warnings is a daunting task, which significantly limits the practicality of the analysis. In this paper, we develop a novel dynamic technique that automatically validates and categorizes such warnings to unleash the power of static memory leak detectors. Our technique analyzes each warning that contains information regarding the leaking allocation site and the leaking path, generates test cases to cover the leaking path, and tracks objects created by the leaking allocation site. Eventually, warnings are classified into four categories: MUST-LEAK, LIKELY-NOT-LEAK, BLOAT, and MAY-LEAK. Warnings in MUST-LEAK are guaranteed by our analysis to be true leaks. Warnings in LIKELY-NOT-LEAK are highly likely to be false warnings. Although we cannot provide any formal guarantee that they are not leaks, we have high confidence that this is the case. Warnings in BLOAT are also not likely to be leaks but they should be fixed to improve performance. Using our approach, the developer's manual validation effort needs to be focused only on warnings in the category MAY-LEAK, which is often much smaller than the original set.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Performance, Reliability, Experimentation

ISSTA '13, July 15–20, 2013, Lugano, Switzerland

Keywords

Memory Leaks, Concolic Testing, Warning Classification

1. INTRODUCTION

Memory leaks are an important source of severe memory errors that can significantly reduce the availability, performance, and security of real-world programs. According to the US-CERT Vulnerability Notes Database [3], 39% of all reported vulnerabilities since 1991 were caused by memory leaks or memory corruption. In an unmanaged language such as C/C++, memory allocation and deallocation needs to be managed explicitly and manually by developers, which can easily lead to memory errors and vulnerabilities. A memory leak is a common memory error that occurs when a dynamically allocated object becomes unreachable. For example, if a developer creates an object (e.g., using `malloc`) but forgets to release it (e.g., using `free`) after it is no longer needed, the memory space consumed by the object cannot be reclaimed and reused, leading to reduced memory resource and diminished program performance.

Research Problem Both static [52, 44, 32, 31, 15] and dynamic program analysis [28, 40, 45, 43, 16] techniques have been attempted to find memory leaks in C/C++ programs. A static analysis often formulates memory leak detection as a source-sink problem, where object allocation sites and their corresponding deallocation sites are considered as the sources and the sinks, respectively. Because a memory leak exists only in certain control flow paths, most static leak detectors are path-sensitive—among all paths that pass an allocation site, those that do not contain its corresponding deallocation site are identified and reported. When such paths are executed, the object(s) created by the allocation site may not be freed, and thus memory leaks may result. A sound static analysis is able to detect all potential memory leaks in a program; in addition, the analysis needs not run the program and thus does not introduce any execution overhead. Due to these desirable properties, static memory leak detectors have gained much popularity in both academic research and real-world software development. Commercial static analysis tools that can detect memory leaks include, for example, HP Fortify [7], Klocwork [2], and Coverity [5]. These tools have been widely used to help real-world developers find leaks and other memory errors.

However, a highly precise static analysis usually cannot scale to large code base. To find an appropriate balance point, analysis accuracy is often sacrificed for scalability, leading to imprecise modeling of many important program properties such as heap objects and path information. In addition, due to the complex pointer arithmetic

Manual Validation Priority

MAY-LEAK > BLOAT > LIKELY-NOT-LEAK > MUST-LEAK

Fixing Priority

MUST-LEAK > MAY-LEAK > BLOAT > LIKELY-NOT-LEAK

Figure 1: Priority comparisons among the four categories.

operations used widely in C/C++ and the extremely large number of paths in a real-world program, a static analysis often ends up reporting a sea of likely warnings with true problems being buried among them, which significantly limits its real-world usefulness [10]. Our experience shows that even a commercial (and mature) static memory leak detector like HP Fortify [7] can report a great number of warnings for a moderate-sized program. Manually inspecting all of the warnings to find true leaks is a daunting task that can be extremely tedious, labor-intensive, and time-consuming.

Our Proposal In this paper, we propose a novel approach that reduces the human validation effort by dynamically classifying statically generated memory leak warnings. This proposed approach takes a report produced by a static leak detector as input and automatically generates test cases (using concolic testing [23, 48])) to validate each warning in the report. Our approach works for any static memory leak detector that reports the following three pieces of information: (1) an allocation site a, (2) a control flow path fragment p (i.e., a set of branches) on which a is located, and (3) a program point e on p (often a function return point) such that a run-time object created by a may become unreachable when the execution passes e. The basic idea of our approach is to *use path information p to direct the concolic testing engine to generate test cases to cover p and then dynamically track each object created by a in the test runs to validate whether it leaks.* Given the fact that almost all existing (path-sensitive) static leak analyses produce these three pieces of information, our approach can be used immediately to improve the usefulness of dozens of off-the-shelf static memory leak detection tools. In this paper, we use HP Fortify as an example frontend. Fortify is a commercial tool used widely in industrial software development. The effectiveness of our approach on Fortify demonstrates its broad applicability in other (research or industrial, prior or future) static leak detectors.

Particularly, the proposed dynamic approach classifies static memory leak warnings into four categories: MUST-LEAK, LIKELY-NOT-LEAK, BLOAT, and MAY-LEAK. A warning is labeled MUST-LEAK if we can generate a test case that triggers the leak; If the leak cannot be found in any generated test runs, we have a high confidence that the warning is a false warning (while we cannot provide any guarantee), and thus, we label it LIKELY-NOT-LEAK. If the dynamic analysis finds that the reported object is released in the end (i.e., not a leak) but it is not used for a long time (i.e., stale), we label it BLOAT. Finally, the warnings that could not be classified in any of these three categories are labeled MAY-LEAK; for these warnings, our approach could not provide any useful information and the developer has to manually validate them. Figure 1 shows comparisons of the leak fixing priority and the validation priority among these four categories. It is easy to see that MUST-LEAK warnings need to be immediately fixed while MAY-LEAK warnings need to be carefully validated. During our experiments, we found MAY-LEAK is often a very small set and validating them may only need a very small amount of manual work. While our technique is designed to validate memory leak warnings, it can be easily adapted to save validation effort for a variety of path-sensitive static analyses.

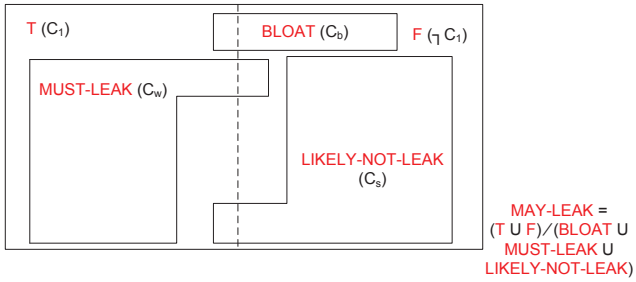

Figure 2: A graphical illustration of the four categories.

Section 2 gives an overview of our technique using concrete examples. Section 3 and Section 4 present, respectively, a formal model of our analysis that explains what our analysis computes exactly and our detailed analysis algorithms. Our experimental results are shown in Section 5. We have run our analysis on the warnings generated by Fortify for 11 programs from various benchmark suites. Among a total of 246 warnings, our analysis is able to automatically classify that 112 are MUST-LEAK warnings, 38 are LIKELY-NOT-LEAK warnings, 38 are BLOAT warnings, and 58 are MAY-LEAK warnings. The number of warnings that need manual validation has been reduced from 246 to 58. Our results also demonstrate, empirically, that warnings labeled LIKELY-NOT-LEAK are highly unlikely to be leaks—we have manually inspected all the LIKELY-NOT-LEAK warnings and have not found any true leak among them.

The major contributions of this paper are as follows:

- A novel classification system for validating static memory leak warnings;
- a novel dynamic analysis algorithm that achieves the classification by combining the path-guided concolic testing and the object-based state tracking;
- an implementation of this technique that uses HP Fortify as the frontend static analysis and the modified CREST concolic testing engine [6] as the test case generation tool;
- a comprehensive evaluation on a set of 11 programs. Our approach has successfully reduced the number of warnings that need to be manually validated by 76.4%. These promising initial results strongly indicate that this technique can be used in practice to improve the usefulness of real-world static leak detection tools.

2. OVERVIEW

This section presents an overview of the proposed approach. We first discuss the general idea of the technique and then use a simple example to illustrate how this technique works.

2.1 Basic Idea

Ideally, the goal of our work is to classify each warning as either a true warning or a false warning, so that the manual validation effort can be completely saved. In this ideal situation, all static warnings can be clearly divided into the true set (denoted by T) and the false set (denoted by F). The F set can be directly discarded while the T set can be sent back to the developers for inspection and fixes. Given a warning with allocation site information a, path information p, and program point e, the warning is a true warning if, in some execution, we can find an object created by a that

```
1  char* foo(int size){
2     char* result;
3     if(size>0){
4        result=(char*)malloc(
                  size*sizeof(char)); // a
5     }
6     if(size>10){
7        return NULL;                 // e
8     }
9     return result;
10}

11 int main() {
12    int i;
13    char *ptr;
14    scanf("%d",&i);
15    ptr = foo(i);
16    if(i > 0 && ptr) {
17       … //use ptr
18       free(ptr);
19    }
20}
```

Fortify Output:
a: line 4; ***p:*** 3T→ 4→ 6T→ 7; ***e:*** line 7

(a) MUST-LEAK

```
1 char* ptrArr[256];                  // global array
2 unsigned int index = 0;             //current index
3 void addToGlobal(char * p) { ptrArr[index++] = p; }
4 void foo(int len) {
5    char *p, fastbuf[10];
6    if (len <= 10)
7       p = fastbuf;
8    else{
9       p = (char *)malloc(len);       //  a
10      addToGlobal(p);                // add p to ptrArr
11   }
12   return;                           //  e
13}

14 int main() {
15    int len, i, j;
16    scanf("%d",&len);
17    for (j = 0; j < 10; j++)  addToGlobal(...);
18    foo(len);
19    for(i = 0; i < index; i++) {
20          write(ptrArr[i]); //use site
21    }
22    freePtrArr();     //free all Objects in ptrArr
23 }
```

(b) LIKELY-NOT-LEAK

```
14 int main(){
15    int len,i, j;
16    scanf("%d",&len);
17    for (j = 0; j < 10; j++) addToGlobal(...);
18    foo(len);
19    for(i = 0; i < 10; i++) {
20          write(ptrArr[i]);
21    }
22    freePtrArr();          //free all objects ptrArr
23 }
```

Fortify Output: ***a:*** line 9; ***p:*** 6F→ 9→ 12; ***e:*** line 12

(c) BLOAT

Figure 3: Code examples, Fortify warnings, and our classifications.

does not have any incoming reference right after e (i.e., condition C_1). Conversely, the warning is a false warning if, in all possible executions, all objects created by a have incoming references after e (i.e., condition $\neg C_1$). However, it can be extremely difficult to precisely distinguish true warnings from false warnings—the unrestricted type conversions and the heavy use of pointer arithmetic in C/C++ make it particularly difficult to understand exactly the number of incoming references for a run-time object. For example, LeakPoint [16] employs a complicated case analysis to propagate reference information for different kinds of arithmetic operations. Doing this requires expensive whole-program instrumentation and data flow tracking. Using the approach to validate hundreds of static analysis warnings may take a considerable amount of time (because the validation of each warning often needs many test runs), leading to reduced usefulness and practicality.

To solve the problem, we relax the requirement (of tracking pointer reference counts). Instead of verifying condition C_1, we use a new condition to *approximate* T. This condition is weaker than C_1 but is much easier to check. Particularly, for a warning, we check whether there exists a test execution in which an object created by the reported leaking allocation site a is *not reclaimed (freed) at the end of the execution* (i.e., condition C_w). This condition is easier to check because the expensive *pointer-based reference tracking* is reduced to the *object-based state tracking*. Tracking object state is much more efficient because, for each warning, there is often a very small number of objects created by its reported leaking allocation site. Only these objects need to be tracked for verifying C_w, while verifying C_1 requires instrumenting all stack/heap read/write statements, and tracking pointer propagation throughout the execution. If C_w holds for the warning, we are certain that the warning is a true leak, although the object may not necessarily lose all its references at point e. This warning is then classified into the category MUST-LEAK.

If C_w does not hold for a warning, it does not necessarily mean that it is a false warning. This is because we may not have sufficient test cases to cover certain control flow paths during validation and a leak may occur on such a path. While the concolic testing engine we employ can achieve a high path coverage, there is no guarantee that all paths containing fragment p can be executed, especially in large programs whose inputs are pointers and memory graphs. It is often challenging for concolic testing to generate such non-trivial program inputs. As a result, it can be very difficult to tell, definitively, whether a warning is a false warning. In order to help the developer further reduce her validation effort, we create a category called LIKELY-NOT-LEAK, based on an important observation that there is often a large set of warnings for which no leaks can be seen in all tests executed during our automated validation. While we cannot provide any guarantee that they are not leaks, we are highly confident that this is the case. In other words, we want to use LIKELY-NOT-LEAK to approximate F.

We use directed concolic testing to generate as many test cases to cover p as possible. More precisely, for a particular warning, if, in all tests executed, all objects created by the reported leaking allocation site a are reclaimed at the end of the execution, we have confidence that the warning is a false warning. To further increase our confidence, an even stronger condition is used for identifying LIKELY-NOT-LEAK warnings, that is, all objects created by a in all our tests (1) are *accessed* after point e and (2) are reclaimed in the end (i.e., condition C_s). Warnings that satisfy C_s indicate that the reported objects are actually *used* after the reported leaking points and thus they are highly unlikely to be leaks. Hence, for the developer, the priority of manually validating LIKELY-NOT-LEAK warnings is very low.

Certain warnings do not satisfy part (1) of C_s. For these warnings, all objects created by a are reclaimed at the end of the execution, but some of them are *never used* after point e (i.e., condition C_b). Although these objects are not leaking, they are *stale* and should be reclaimed earlier for improved performance. Such warnings are classified into the category BLOAT. Inspecting and fixing problems in BLOAT is not urgent. It can be done only when clear performance degradation is seen.

If no test case can be generated to cover p (e.g., the path constraints are too difficult to solve), the warning is put into the category MAY-LEAK. We neither can prove it is a leak nor have confi-

dence that it is not a leak. The developer's validation priority should be given to these warnings in order to fully understand them. Our experience shows that this set is often very small and, therefore, the developer's validation effort can be significantly reduced. A graphical illustration of these categories is given in Figure 2. Sets T and F are separated by a dash line. MAY-LEAK is represented by the area not included in any of the three categories. Note that different categories may have intersections. Details about these intersections will be explained later in Section 3.

2.2 Example

Figure 3 shows three code examples, their warnings generated from Fortify, and our classifications. Function `foo` in Figure 3 (a) creates an object at line 4. Fortify reports that this allocation site may create a leaking object. The path fragment p in the warning is $3T \rightarrow 4 \rightarrow 6T \rightarrow 7$ and the leaking point e is line 7. To validate this warning, we first instrument the program by inserting code at line 4, line 7, and the end of `main` to track the usage of the object(s) created by a. Particularly, for each such object, we create a piece of *tracking data*, which records the run-time state of the object. It is updated once an important event (e.g., allocated, used, or released) occurs to the object. Detailed discussion of the state update can be found in Section 4. The tracking data for each tracked object is inspected at the end of the execution to identify whether the object is released and whether it is used after the leaking point.

We modify CREST, a concolic testing engine for C, to generate tests for the validation. Path information p is used to direct CREST to efficiently generate tests to cover p. In particular, using an interprocedural control-dependence analysis, we pre-compute a path reachability table that contains, for each control-flow branch in the program, a boolean flag indicating whether p can be potentially reached from this branch. During the test generation, we do not generate (and solve) constraints for a branch if its flag is false. Hence, only relevant test cases that can drive the execution to p are generated and run, leading to increased efficiency in both test case generation and test executions. For our example in Figure 3 (a), a test input $len = 11$ is generated. During the execution of the test, the object created at line 4 is tracked. At the end of the execution, we find that the object is not released. We therefore conclude that this warning is a MUST-LEAK warning because a leaking evidence has been found.

Figure 3 (b) and (c) show a LIKELY-NOT-LEAK warning and a BLOAT warning, respectively. The two code examples differ only in the loop variable at line 19. Fortify reports the same warning for both programs. The warning suggests that the leak may occur in function `foo`: the object created at line 9 is added into a global array (declared in line 1), which makes Fortify determine that the references of the object will be lost after `foo` returns. However, in the end, all objects in the global array are freed (at line 23). Note that such an object usage is very common, because a real-world program often makes heavy use of user-defined data structures (such as lists and maps). The imprecise handling of such data structures can significantly limit the usefulness of Fortify (and other similar static leak detectors).

Similarly, CREST generates a test input $len = 11$ to exercise the reported path. The object created at line 9 is tracked during the test execution. Note that when the object is added to the array `ptrArr` at line 10, `ptrArr` already contains 10 elements (due to the call at line 17). Hence, the object's index in `ptrArr` is 10. In Figure 3 (b), the object is retrieved from the array and accessed at line 20, because the loop (at line 19) traverses the entire array. The tracking data of the object is updated to a special state ("Used") upon the access of the object at line 20 and then updated to "FreedAfterUsed" when it is freed at line 22. At the end of the execution, we see that the object is used after the leaking point (line 12) and is eventually released, and thus, we put it into the category LIKELY-NOT-LEAK.

If the loop variable in line 19 is changed to 10 (shown in Figure 3 (c)), the warning will be classified as BLOAT. Although the object created at line 9 is released in the end, it is never used after the execution passes line 12. This is because the loop (at line 19) traverses only the first 10 elements in the array and thus this object is not accessed. While this warning is not a leak, it points to an inefficient implementation: if an object is no longer needed after a certain point, it should be freed right away. Warnings labeled BLOAT in our work essentially correspond to memory leaks in managed languages [56, 19, 41, 11, 33, 46, 29, 12, 13, 51, 55]—although the virtual machine guarantees that such objects will be eventually reclaimed, the accumulation of these objects can cause severe performance degradation and even program crashes. If the SAT solver determines that p is an infeasible path, we classify the warning also as LIKELY-NOT-LEAK. Only warnings for which no test case can be generated are classified into the category MAY-LEAK.

3. FORMAL DESCRIPTION

This section presents a lightweight formalism to define the four categories and explains how warnings are classified by our analysis. We first define a simple imperative language (a core subset of C) for the formal development:

Variables	$u, v \in \mathsf{V}$
Labels	$l \in \mathsf{L}$
Integers	$n \in \mathsf{I}$
Predicates	$\phi \in \Phi$
Statements	$e \in \mathsf{E}$

$$e ::= u =^{l} v \mid u =^{l} malloc(n) \mid u =^{l} *v \mid *u =^{l} v \mid {}^{l} free(u) \mid e\,;e \mid \text{if } (\phi) \text{ then}^{l^T}\ e \text{ else}^{l^F}\ e$$

This language contains all important features of C that are necessary for us to validate a memory leak warning. Each statement e has a label $l \in \mathsf{L}$ that is used to identify e. For simplicity of the presentation, function calls are not included in this language. They can be thought of as being completely inlined into `main`. Based on this language, we first give the definition of static leak warning as follows:

Definition 1 *(**Static Leak Warning**) A static warning w is a triple $(a,\ p = \langle l_1^{b_1}, l_2^{b_2}, \ldots, l_n^{b_n}\rangle,\ e)$ where $a, e, l_1, l_2, \ldots, l_n \in \mathsf{L}$, and $b_i \in \{T, F\}$. a and e are labels for a malloc statement and for a statement at which an object created at a is considered to lose all its references by the static analysis, respectively. Labels $\langle l_1^{b_1}, l_2^{b_2}, \ldots, l_n^{b_n}\rangle$ identify a chain of branches, forming a path fragment containing a and e.*

In most cases, $\langle l_1^{b_1}, l_2^{b_2}, \ldots, l_n^{b_n}\rangle$ represents a path fragment that begins at a and ends at e. Using ϕ_i to denote the predicate that guards l_i, the leak validation test suite is defined as:

Definition 2 *(**Leak Validation Test Suite**) Given a warning $w = (a,\ p = \langle l_1^{b_1}, l_2^{b_2}, \ldots, l_n^{b_n}\rangle,\ e)$, a leak validation test suite S_w for w contains a set of test cases $(t_1, t_2, \ldots, t_n)$ such that the execution of each test case t_i satisfies the path constraint $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$, where $\varphi_i = \phi_i$ if $b_i = T$; $\varphi_i = \neg\phi_i$ otherwise.*

We say a leak validation test suite S is complete if for each possible CFG path that contains fragment p, there exists a test case

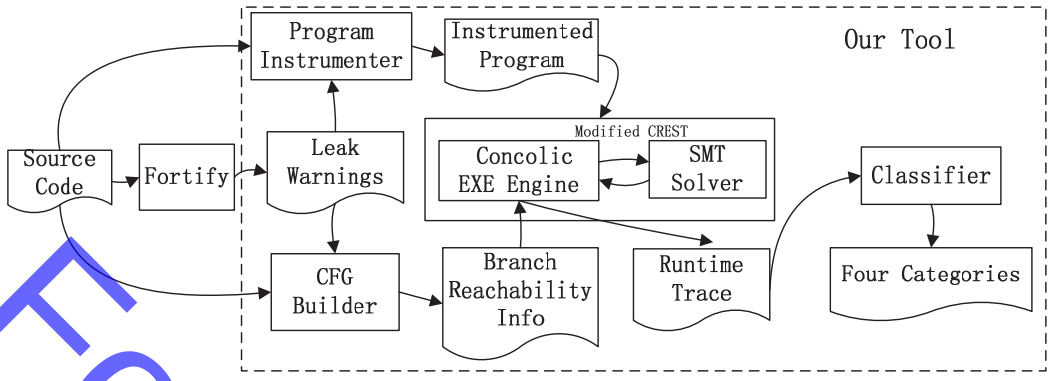

Figure 4: An overview of our tool.

$t \in \mathsf{S}$ whose execution can cover the path. We use concolic testing to generate as many test cases as possible to achieve high coverage. However, there is no guarantee that S can be complete.

We use $\mathsf{O}_{a,t}$ to represent the set of run-time objects created by the allocation site labeled a under test t. For each object $o \in \mathsf{O}_{a,t}$, auxiliary boolean functions $\mathrm{hasIR}(o, e)$, $\mathrm{isUsed}(o, e)$, and $\mathrm{isFreed}(o)$ return, respectively, whether the object has incoming references right after the leaking point e, whether the object is used after e, and whether it is finally reclaimed. A static leak warning w $(= (a, p, e))$ is a *true warning* if we can generate a test suite S_w for w such that the following condition (C_1) holds.

$$\exists t \in \mathsf{S}_w\colon \exists o \in \mathsf{O}_{a,t}\colon \neg\mathrm{hasIR}\,(o, e) \qquad (\mathsf{C}_1)$$

As described earlier in Section 2.1, it can be very difficult to precisely compute hasIR for each object, and thus, we use the following (weaker) condition (C_w) to approximate:

$$\exists t \in \mathsf{S}_w\colon \exists o \in \mathsf{O}_{a,t}\colon \neg\mathrm{isFreed}\,(o) \qquad (\mathsf{C}_w)$$

Definition 3 ***(MUST-LEAK)*** *A warning w is a MUST-LEAK warning if its test suite S_w satisfies C_w.*

Because C_w is weaker than C_1, MUST-LEAK (determined by C_w) may contain false warnings. For such a warning ($\in$ MUST-LEAK $\cap$ F in Figure 2), its reported object is indeed leaking, but the point at which it loses its references is later than the reported leaking point e. Fixing these warnings is necessary: they are true leaks even if the reported path information and leaking point are problematic.

It is important to note that if S_w is a complete test suite for w, we would not even need to actively look for false warnings—w is guaranteed to be a false warning if w does not satisfy C_w. However, all-paths test generation for general programs with aliases has been shown to be NP-hard [36], and it is also impossible to know whether S_w is complete if p involves complex program structures such as loops and recursion. To approximate F, we use the following condition (C_s) to find likely false warnings:

$$\forall t \in \mathsf{S}_w\colon \forall o \in \mathsf{O}_{a,t}\colon \mathrm{isUsed}\,(o, e) \wedge \mathrm{isFreed}\,(o) \qquad (\mathsf{C}_s)$$

Definition 4 ***(LIKELY-NOT-LEAK)*** *A warning w is a LIKELY-NOT-LEAK warning if (1) the path constraint $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n \equiv$ false or (2) its test suite S_w satisfies C_s.*

First, if the path constraint generated for p is guaranteed to result in false (which indicates that p is an infeasible path), w must be a false report and, therefore, the warning is classified as LIKELY-NOT-LEAK. w can also be classified as LIKELY-NOT-LEAK if, in all test runs generated for the warning, all objects created by the reported allocation site are used after the reported leaking point and are eventually freed. A small portion of LIKELY-NOT-LEAK warnings may be true leaks (i.e., LIKELY-NOT-LEAK $\cap$ T in Figure 2), because these warnings may exhibit leaks in certain paths that have not been covered by our tests. However, we have not found any warnings in this intersection in our experiments, either. On the other hand, a false warning may not be classified as LIKELY-NOT-LEAK if the reported object is freed before the reported leaking point e (and hence it is never used after e). This situation is also very rare, because a static analysis is often conservative and e should always be the earliest program point at which the static analysis determines that the object may lose all its references. The following condition (C_b) defines the category BLOAT:

$$\forall t \in \mathsf{S}_w\colon (\forall o \in \mathsf{O}_{a,t}\colon \mathrm{isFreed}\,(o)) \wedge (\exists o \in \mathsf{O}_{a,t}\colon \neg\mathrm{isUsed}\,(o, e)) \qquad (\mathsf{C}_b)$$

Definition 5 ***(BLOAT)*** *A warning w is a BLOAT warning if its test suite S_w satisfies C_b.*

Similarly, BLOAT may contain true warnings (i.e., BLOAT $\cap$ T in Figure 2) due to the insufficient path coverage.

Definition 6 ***(MAY-LEAK)*** *A warning w is a MAY-LEAK warning if (1) the path constraint $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n \not\equiv$ false and (2) $\mathsf{S}_w = \emptyset$.*

MAY-LEAK contains all the warnings such that their leaking path p may be feasible but no test case can be generated to exercise it. These warnings cannot be classified into any of the previous three categories and careful manual inspection needs to be performed to validate them. For each warning w in LIKELY-NOT-LEAK, it is clear to see that the higher path coverage the test suite S_w can achieve, the more confident we are that w is not a leak. Hence, an interesting further step would be to rank warnings in LIKELY-NOT-LEAK based on the quality (e.g., coverage) of their test suites S_w. In our work, however, we do not see a strong motivation to do so, because we have not found any true warnings that are mistakenly classified as LIKELY-NOT-LEAK in our experiments.

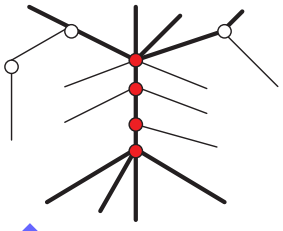

Figure 5: A test generation example; each circle represents a conditional; the highlighted circles form the path fragment p and the highlighted lines show the paths that we intend to generate tests to cover.

4. ANALYSIS ALGORITHMS

Tool Overview Figure 4 shows an overview of our tool. Given a static leak warning $w = (a, p, e)$, two pre-processing phases are performed prior to the test case generation. In the first phase, the target program is instrumented. The inserted code serves two major purposes: (1) it declares symbolic variables and marks the path fragment p in the source code for the subsequent path-directed concolic testing; (2) it tracks the usage of each run-time object created by the reported allocation site a and updates its tracking data. The second phase consists of a path reachability analysis, which is performed on the CFG of the program to determine, for each control flow branch, whether an execution following the branch could potentially reach each branch on p. This analysis is straightforward: it combines an intraprocedural control-dependence analysis with an interprocedural call graph traversal. The concolic testing engine is then modified to be aware of this reachability information so that the test case generation is guided to explore only the paths that may reach p, leading to a reduced search space in the symbolic execution. The tracking data for each tracked object is inspected at the end of the execution to classify the warning.

Path-Guided Concolic Testing Algorithm 1 shows the algorithm for the path-guided test generation. The algorithm is implemented by modifying CREST [6], an existing concolic testing tool for C. The input of the algorithm is the path fragment p of a warning and a reachability map m (pre-computed by the reachability analysis). The basic idea of the algorithm is to use m to direct CREST to generate only the tests that can execute p. A graphical illustration of the paths we want to generate tests to cover is given in Figure 5.

p is essentially a list of branch labels $l_0, l_1, \ldots, l_{k-1}$ reported by the static analysis (i.e., Fortify). m maps each branch t in the program to a reachability vector $\langle b_0, b_1, \ldots, b_{k-1}\rangle$. Each b_i is a boolean flag indicating whether t may reach branch l_i on p at run time. Note that if t can reach branch l_i on p, it must also reach all its following branches $l_{i+1}, l_{i+2}, \ldots, l_{k-1}$ (assuming that p is a feasible path). We first run the target program with an initial set of test cases (lines 3–5). The concolic execution of the program (line 5) records all the branches that have been covered during the run in B. For each branch label in B, the concolic testing engine also records its corresponding predicate (expressed in terms of the input symbolic variables) in C. The conjunction of the symbolic predicates in C forms the path constraint for the current execution. The original concolic testing algorithm (e.g., proposed in [48, 23]) generates new path constraints by negating the symbolic predicate for each branch in the current execution. These new constraints are solved to generate test cases to cover the alternative paths. While a similar approach is employed in our work, we generate constraints only for paths that may reach p.

Algorithm 1 Path-Guided Test Input Generation

```
Input:
    The path fragment p : ⟨l_0, l_1, ..., l_{k-1}⟩
    Map m : ⟨l, ⟨b_0, b_1, ..., b_{k-1}⟩⟩ // computed by the reachability analysis
 1: Array B ← {} //A branch label list
 2: Array C ← {} //A symbolic predicate list
 3: I ← InitTestInputs()
 4: /*concolic execution of the program with an initial test suite I*/
 5: Run(I, &B, &C)
 6: PathGuidedSearch(B, C) //search for new test inputs
 7:
 8: Procedure PathGuidedSearch (Array B, Array C)
 9: int count ← 0
10: bool reachable ← TRUE
11: int n ← length(B) - 1
12: for i ← 0 to n do
13:     if count < k then
14:         reachable ← m(B̄[i])[count]
15:     end if
16:     if reachable or count = k then
17:         I' ← solve(C[1] ∧ C[2] ∧ ... ∧ C[i − 1] ∧ ¬C[i])
18:         B' ← {}
19:         C' ← {}
20:         Run(I', &B', &C')
21:         PathGuidedSearch(B', C')
22:     end if
23:     if B[i] = l_count then
24:         count ← count + 1
25:     end if
26: end for
```

Using lists B and C, procedure *PathGuidedSearch* in line 6 generates and solves new path constraints. Each iteration of the main loop (lines 12–26) inspects a branch label $B[i]$, and attempts to generate test cases to cover its alternative branch (denoted as $\overline{B[i]}$). This is done by consulting with map m (line 14). We use a variable $count$ to track how many branches on p have already been reached on the current execution path (lines 23–25). The initial value of $count$ is 0 (line 9). Before the first branch l_0 is reached, we always check whether branch $\overline{B[i]}$ can reach l_0 (line 14). Note that the expression $m(\overline{B[i]})[count]$ returns a boolean value indicating whether $\overline{B[i]}$ can reach the $count$-th branch on p (i.e., l_{count}). If it can, we negate its corresponding symbolic predicate $C[i]$ to form a new path constraint (line 17). This constraint is sent to a SAT solver (i.e., Yices [4] in our work) in order to generate test inputs to cover $\overline{B[i]}$.

If $B[i]$ and l_0 are the same label, $count$ is incremented (line 24), and hence, from the next iteration of the loop, we will check if branch $\overline{B[i+1]}$ can reach (the next) branch l_1 on p. If, in a certain iteration j of the loop, we find that all the branches $l_0, l_1, \ldots, l_{k-1}$ on p have been reached in the current execution (i.e., $count = k$), all the subsequent iterations of the loop will no longer check branch reachability. All branches $B[h]$ and $\overline{B[h]}$ $(h > j)$ will be reachable from p, and thus, we will generate test cases to cover all of them to track the reported objects after the leaking point e.

Update Tracking Data A piece of tracking data is maintained for each object created by the reported allocation site. There are multiple ways to maintain tracking data. For example, a shadow heap [54, 42, 53] can be created so that each heap location has a corresponding shadow location to store its tracking data. An alternative approach [8] is to use a hash map to store tracking data for each tracked object. We employ the hash-map-based approach

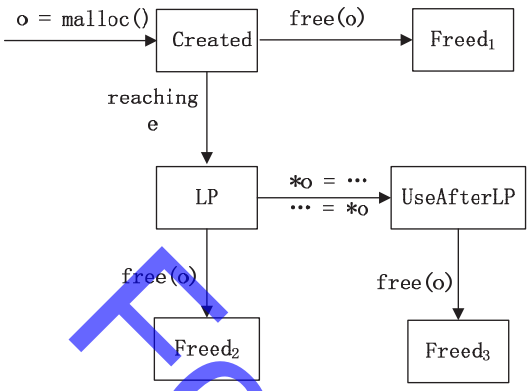

Figure 6: The state machine of a tracked object.

in this work, because only a small number of objects need to be tracked and it is thus not worth shadowing the entire heap.

The tracking data of each tracked object records the state of the object. Figure 6 shows the state machine. The initial state of each object is "Created", and it transitions to "LP" when the execution reaches the reported leaking point e. Upon the use of the object (after e), its state becomes "UseAfterLP". There are three "Freed" states. The object reaches "Freed$_1$" if it is freed before point e. As described in Section 3, this case is extremely rare because the static analysis is unlikely to report such warnings. State "Freed$_2$" is reached if a `free` call is executed on an object in state "LP". Objects in this state are likely to be bloat—the `free` call should be made earlier to free them (immediately after they are no longer used). The object will reach "Freed$_3$" if it is used after the leaking point and eventually freed. At the end of the execution, the instrumentation code inspects tracking data for all tracked objects to determine the results of auxiliary functions isUsed and isFreed. isFreed(o) returns true if o is in any of the three "Freed" states. isUsed(o, e) returns true if o's tracking data is either "UseAfterLP" or "Freed$_3$". These functions are then used to classify warnings (as described in Section 3).

5. EVALUATION

Implementation Our instrumentation was performed using the CIL[1] instrumentation framework. The CREST [6] concolic testing engine was modified to perform the path-guided test case generation. The static memory leak detector used in our experiments was HP Fortify version 3.2. All experiments were executed on a quad-core machine with an Intel Core (TM) i5-3360M 2.80GHz processor, running Linux 3.2.0.

We performed two experiments to evaluate the effectiveness of our validation approach. The first experiment is designed to assess the precision of the classification technique and the second experiment is designed to understand whether our approach can scale to large applications. The goal of these experiments is to answer the following four research questions:

- **RQ1**: How accurate is our classification system?
- **RQ2**: How much effort can a developer save by validating static memory leak warnings using our technique?
- **RQ3**: How efficient is our technique?
- **RQ4**: How does our technique perform on large-scale, real-world applications?

Experimental Subjects We perform our first experiment using a set of programs from the Siemens [27] (i.e., replace–tcas in Table 1) and the coreutils [1] benchmark suites (i.e., the rest of the programs in the table). These programs are relatively small and it is thus easy for us to understand their implementation logic to manually validate our classification results. To increase the number of static analysis warnings for each program, we manually injected both true and false leaks. The second experiment includes a case study on the use of our tool to validate leaks for a large-scale application (i.e., *texinfo-4.33*). No leak injection was done for this application.

5.1 Experiment 1: Classification Accuracy and Efficiency

For each program in Table 1, both true and false leaks were injected. Injecting true leaks was done by randomly removing calls to `free`. It is much more difficult to inject false leaks, because whether or not the injected code can be detected as a leak depends on the precision of the static analysis. As we did not have access to Fortify's implementation, we attempted a few different coding patterns and ended up finding one pattern that could direct Fortify to report false warnings. Particularly, we declared a global map and added dynamically-allocated objects into the map. The entire map is released at the end of execution (e.g., an example has been shown in Figure 3). We inserted this code pattern at random places in each program. To create BLOAT cases, we added code to retrieve a random number of elements in a global map before it is released. If an object in the map is visited in the traversal, it should be labeled LIKELY-NOT-LEAK; otherwise, it should be labeled BLOAT. Manual injection of true/false leaks makes it easier for us to assess the accuracy of our classification system—leak warnings are created purposefully for different categories, and it is immediately clear whether or not our tool does the appropriate classification by inspecting the classification results.

All the experimental data for these 10 programs is reported in Table 1. Section (a) shows, for each program, the number of lines of code (#LOC), the number of static analysis warnings Fortify outputs (#W), and the number of test cases generated by our path-guided test generation approach (#S). Section (b) shows our classification results (i.e., #Must for MUST-LEAK, #LNL for LIKELY-NOT-LEAK, #B for BLOAT, and #May for MAY-LEAK). Section (c) reports the numbers of warnings in the intersections between various sets. Set T represents the set of true leaks including those that exist in the original program and those that are injected by ourselves. Hence, #(LNL ∩ T) and #(B ∩ T) represent, respectively, the numbers of true leaks that are mistakenly classified into LIKELY-NOT-LEAK and BLOAT. Section (d) reports our time and space overhead measurements. T_0 and T_1 show, for each program, the running times of executing all the tests without and with the object state tracking, respectively. Similarly, Sp_0 and Sp_1 show the peak memory consumptions without and with the object state tracking, respectively.

Our first observation is that, for each program, both (LNL ∩ T) and (B ∩ T) are empty sets. This indicates that our approach has not falsely classified any true leak and is thus highly precise. Our detailed studies of the warnings suggest that the leaky behavior of a true leak usually manifests quickly when its path fragment p is exercised in a test. Very often, a small number of test cases generated to cover p are sufficient to reveal whether or not the warning is a true leak. This observation confirms our assumption that if no leaky behavior is seen in all tests generated for a warning, we have strong

[1]http://sourceforge.net/projects/cil/.

Table 1: Categorized warnings: reported in Section (a) are lines of code (#LOC), numbers of warnings reported by Fortify (*#W*), and execution frequencies of the leaking paths during path-guided concolic testing (*#S*); Section (b) reports numbers of warnings in each of the four categories: MUST-LEAK (*#Must*), LIKELY-NOT-LEAK (*#LNL*), BLOAT (*#B*), and MAY-LEAK (*#May*); Section (c) reports numbers of true leaks falsely classified into LIKELY-NOT-LEAK ($LNL \cap T$) and BLOAT ($B \cap T$), respectively; Section (d) reports our overhead measurements including running times (T_0 and T_1) and peak memory consumptions (Sp_0 and Sp_1).

Program	(a) General Info			(b) Categorized Warnings				(c) False Classifications		(d) Performance			
	#LOC	#W	#S	#Must	#LNL	#B	#May	#(LNL ∩ T)	#(B ∩ T)	T_0 (s)	T_1 (s)	Sp_0 (Mb)	Sp_1 (Mb)
replace	563	18	3444	5	3	4	6	0	0	3.82	3.95	16.4	20.4
print_tokens	726	22	17383	8	4	6	4	0	0	3.7	5.0	16.9	20.6
print_tokens2	569	29	16943	8	7	9	5	0	0	4.2	4.7	22.1	22.2
tcas	173	8	54	1	4	1	2	0	0	0.06	0.07	15.6	19.8
wc	802	8	6000	2	2	2	2	0	0	10.5	15.5	17.1	22.8
cat	785	8	4002	2	1	2	3	0	0	16.9	26.2	15.7	19.9
head	1063	18	5007	4	6	2	6	0	0	12.2	17.4	16.3	20.5
tr	1949	32	37281	11	8	8	5	0	0	21.2	26.5	16.8	21.2
expand	433	6	3854	1	1	2	2	0	0	22.3	26.9	21.8	25.9
unexpand	535	6	3996	1	1	2	2	0	0	25.7	26.2	23.1	27.2

confidence that the warning is a false warning. As such, the developer does not need to put much effort in inspecting leaks classified as LIKELY-NOT-LEAK and BLOAT.

Our second observation is that the MAY-LEAK category (shown in column #May) is often very small. Our tool has successfully reduced the number of warnings that need to be manually validated by 76.1% (from 155 to 37). We have also inspected all the MAY-LEAK warnings. We found that many warnings are classified into this category because the injected code patterns causing them are in control-flow regions that can only be executed by very complicated test inputs. Generating test cases that trigger these regions is clearly beyond the capabilities of the existing concolic testing techniques. Among these 37 MAY-LEAK warnings, 13 warnings are true leaks. These leaks can only be found by manual validation.

Our third observation is that the proposed technique (that combines path-guided concolic testing and state-based object tracking) is efficient. The execution of thousands of test cases for each program takes only a very small amount of time and memory space. The overall running time and space overheads incurred by the object state tracking are 24.8% and 21.4%, respectively. Note that these low overheads result primarily from the use of condition C_w to approximate C_1—had the pointer-based reference tracking been employed, the overhead would have increased to more than 100× (e.g., the pointer-tracking approach used in LeakPoint [16] incurs a 100–300× time overhead).

5.2 Experiment 2: Scaling to a Large-Scale Application

The second experiment evaluates the scalability of our technique. We focus on a large-scale C program, called *texinfo*[2], version 4.13. *texinfo* is a text format transformation system that uses a single source file to produce output in a number of formats, both online and printed (dvi, html, info, pdf, xml, etc.). It has a total of 46493 lines of code. The input of this application has two components: a file component that specifies the source file, and a command line component that specifies ways to transform the file to generate output. It can be extremely difficult for concolic testing to automatically generate the file input. To solve the problem, we manually wrote a source file for each acceptable input format, and let CREST, our concolic testing engine, automatically generate the command line input that chooses from this set of files to test the program. It took us about 1.5 weeks to write these input files. Although this work was done manually, it is not a major limitation of our approach. This is because this need to manually write test cases is due to the inability of concolic testing, which is not our contribution; one can think of this set of inputs as the initial test cases needed to drive concolic testing. In addition, typical real-world programs often contain such inputs (used by their own developers), which can be directly used to run our tool. In fact, we found that this test generation approach combines the advantages of concolic testing and manual test development. On one hand, it uses the developer's insight to overcome the difficulty of the concolic testing engine in generating tests with complicated structures, such as texts with specific formats. On the other hand, it exploits concolic testing's path exploration to achieve high coverage for the command line input, which can often be difficult for human developers to achieve manually.

Fortify reports a total of 91 warnings for *texinfo*. Our analysis has successfully classified 70 of them. For the rest (21) of the warnings, no test case can be generated by CREST to exercise their reported path fragments and thus they are classified as MAY-LEAK. This is primarily because the path constraints for these warnings are too complicated to solve. We expect that the size of this set will significantly decrease in the future as more powerful solvers become available. Among the 70 warnings that are precisely classified, the numbers of warnings in MUST-LEAK, LIKELY-NOT-LEAK, and BLOAT are, respectively, 69, 1, and 0. We have carefully inspected all the MUST-LEAK warnings and found that, among the 69 warnings, 59 were true leaks and the remaining 10 warnings were due to the use of global pointers. These objects are referenced in global data structures and are never freed during the program execution. According to condition C_1, the path information contained in these warnings is imprecise because the reported objects still have references after the execution passes the leaking points. However, such warnings reveal *inefficiencies* in the program that need to be fixed for better performance—the accumulation of a large number of unnecessarily cached objects can easily lead to performance degradation and even program crashes; developers should reclaim them immediately after their last use.

We manually verified that the only LIKELY-NOT-LEAK warning is not a leak. Despite the many limitations of the concolic testing engine we use, the proposed technique has successfully reduced the number of warnings that need to be manually validated by 76.9%. The time and space overheads for running this application are 77.5% (from 10.7s to 19.0s) and 26.4% (from 15.9Mb to 20.1Mb), respectively.

[2]http://www.gnu.org/software/texinfo/.

5.3 Discussion

This subsection presents a discussion of our studies in order to draw more general conclusions. In general, we found that our technique is useful in reducing the amount of manual effort to validate static analysis warnings. When examining the results from Experiment 1, we found that for all programs in our benchmark set, our technique could accurately classify their static analysis warnings as long as the path fragments in these warnings are exercised in concolic testing. Furthermore, only a small number of warnings could not be automatically validated. As such, for RQ1 and RQ2, *we assess that our classification system is accurate and significant manual effort could be saved by validating static leak warnings using our approach.*

In terms of analysis efficiency, we collected performance measurements before and after our technique was used. When comparing these measurements, we found that our technique incurs very small time and space overheads. As such, for RQ3, *we assess that our technique is efficient and our approximations are effective. The proposed approximations have led to a significantly more efficient implementation and yet have not introduced any mistreatment and imprecision.*

In terms of analysis scalability, we evaluated our technique on *texinfo*, a large, real-world application. We found that our technique could precisely classify most of static leak warnings and the manual validation of all classified warnings shows that no true leak has been falsely classified. As such, for RQ 4, *we assess that our technique can scale to large applications. Its scalability and the effectiveness may further increase with the future advancement of the SAT technology.*

In summary, our experimental results suggest that the use of our dynamic technique to classify static leak warnings provides an effective way to automatically valid these warnings in order to make static memory leak detection more feasible and powerful for real-world use.

Threats to Validity Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the accuracy of our classification system on a limited set of programs, and thus we are unable to definitively state that our technique will be accurate for real-world programs in general. However, we are confident that these results are indicative of the impacts of using the technique to validate warnings for various kinds of real-world C/C++ programs.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. Our experiments measured the reduction in manual validation effort only in terms of the size of the MAY-LEAK category. Although these results give an indication of numbers of warnings that do not need to be manually validated, they do not actually show how these numbers will affect either the developer time or the difficulty of manually validating these warnings. Further studies that involve users with different levels of programming skills and experience will be needed to understand the actual developer time saved by using our automated validation system.

6. RELATED WORK

In this section, we focus our discussion on three major types of techniques that are closely related to the proposed work: static and dynamic memory leak detection, directed test case generation, and techniques for ranking and pruning static leak warnings.

6.1 Static Memory Leak Detection

Static analysis has been widely used to find memory errors such as double frees and missing frees for C/C++ programs. Work from [15] reduces the memory leak analysis to a reachability problem on the program's guarded value flow graph, and detects leaks by identifying value flows from the source (malloc) to the sink (free). Saturn [52], taking another perspective, reduces the problem of memory leak detection to a boolean satisfiability problem, and uses a SAT-solver to identify potential bugs. Dor et al. [21] propose a shape analysis based on 3-valued logic, to prove the absence of memory leaks in several list manipulation functions. Hackett and Rugina [26] use a shape analysis that tracks single heap cells to identify memory leaks. Orlovich and Rugina [44] propose an approach that starts by assuming the presence of errors, and performs a backward dataflow analysis to disprove their feasibility. Clouseau [31] is a leak detection tool that uses pointer ownership to describe variables responsible for freeing heap cells and formulates the analysis as an ownership constraint system. Work [32] proposes a type system to describe the object ownership for polymorphic containers and uses type inference to detect constraint violations. Recent work [49] uses a full-sparse value-flow static analysis for leak detection. In addition to the effort in the research community, commercial tools such as Coverity [5], Fortify [7], and Klocwork [2] have gained popularity in the real-world software development. However, it is well-known that all static approaches are limited by the lack of general, scalable, and precise reference/heap modeling. Despite a large body of work on such modeling, it remains an open problem for analysis of large real-world systems. Our technique is orthogonal to all existing static detection techniques—it is designed to prune static analysis warnings to improve the usefulness of (existing or future) static memory leak detectors.

6.2 Dynamic Memory Leak Detection

Purify [28] pioneers the work of dynamic memory leak detection. Other dynamic memory leak detectors for C/C++ programs include SafeMem [45], LeakPoint [16], and Omega [40]. SafeMem is a low-overhead technique that detects memory leaks and memory corruption on-the-fly by exploiting the ECC memory. LeakPoint uses a taint-based approach to track object usage in order to precisely locate the location where an object loses it references and its last use site. Omega (that is similar in spirit to the technique proposed in [39]) also maintains a pointer count for each area of allocated memory. Unlike LeakPoint that tracking the reference flow, Omega intercepts all writes to memory to identify pointers and update the pointer count for each object. Our technique is fundamentally different from these dynamic analysis techniques in the following two important aspects. First, the goal of our technique is to use dynamic analysis to validate a static analysis warning, instead of running the program to find leaks. Thus, there exists clear target information that can be exploited in our analysis while existing dynamic analyses do not have such information. Second, because we are interested in only a very small number of objects for each warning, it is not worth doing whole-program instrumentation and dataflow tracking. Our tool adopts novel approaches that can efficiently track object usage and yet achieve high precision in classifying static analysis warnings. Dynamic techniques are widely used to detect memory leaks in managed languages, such as Java [56, 19, 41, 11, 33, 46, 29, 12, 13, 51, 55]. As described in Section 2, Java memory leaks correspond to the category of BLOAT in this work. Although our tool can also validate such leaks, there does not exist any practical static analysis that can find leaks in managed languages. This is primarily because finding bloat requires precise liveness analysis, which often cannot scale to real-world applications.

6.3 Directed Test Generation

The Synergy algorithm [24] combines model-checking and DART [23] to attempt to cover all abstract states of a program. Our tool is not intended to achieve high coverage, but instead it exploits concolic testing as a component of the classification system. DSD-Crasher [17] is a tool that performs dynamic, static, and dynamic analysis in sequence to find bugs, while our technique automatically generates test cases to validate memory leak warnings. Zhang et al. [57] propose a combined static and dynamic approach to automatically generate legal and diverse method call sequences for unit testing. Babić et al. [9] propose a technique that uses static analysis to guide dynamic automated test generation for binary programs. Taneja et al. [50] use path-based test generation to produce efficient regression tests. Cui et al. [18] propose a rule-directed symbolic execution technique to efficiently check systems rules. Ge et al. [22] develop DyTa, a tool that combines static verification and dynamic test generation to reduce false positives and achieves efficiency. While the core algorithms of these techniques are similar in spirit to the path exploration algorithm in our work, test case generation is just one building block of our approach, and our major contribution is a new classification system that can automatically validate memory leak warnings.

6.4 Ranking Static Analysis Warnings

A practical problem challenging the use of static analysis tools is that it reports a large number of false positives. Many techniques have been developed to address this problem. Ruthruff et al. [47] develop statistical models to predict the foregoing types of warnings from signals in the warnings and implicated code. Heckman et al. [30] propose a false positive mitigation technique and a benchmark, FAULTBENCH, to evaluate the technique. Kim and Ernst classify warnings by analyzing multiple versions of the software [35]. FEEDBACK-RANK, proposed by Kremenek and Engler, is a probabilistic ranking technique that exploits the correlation among reports to reduce numbers of false positives [37]. Z-Ranking prioritizes warnings using the frequency of review results [38]. Boogerd and Moonen employ an execution likelihood analysis to rank warnings [14]. By exploiting the relationship between warnings and bug fixes in the software change history, Kim and Ernst propose a history-based warning prioritization (HWP) algorithm, which ranks warnings using previous warning fix history [34]. Dillig et al. propose an algorithm based on abductive inference to compute the smallest and most general abductions. It can be used to identify the information missed by static analysis to diagnose the reported errors [20]. Clarify is a tool [25] created to improve the error reporting by learning the behaviors of an application based on the summary of its execution history.

Note that all these existing techniques except [20] classify warnings based solely on the information contained in the warnings. They do not automatically validate them using the actual program executions. Work from [20] has a different goal, which is to identify the missing information that can help a developer make better sense of a warning. Despite this large body of work, validating whether a warning is a true or a false warning remains to be a task performed manually by developers. Our approach is designed to bridge this gap—it can be used together with these existing ranking algorithms to not only classify warnings but also validate them.

7. CONCLUSIONS

Static analysis has been widely used to find memory-related problems in the real-world software development. While it can find all potential bugs in a program, a static analysis often reports a great number of false warnings that have to be manually validated by human experts. Validating these warnings is an often a daunting task, which can be extremely time-consuming and labor-intensive. This paper presents a novel dynamic technique that can automatically validate static memory leak warnings for C/C++ programs. Warnings are classified into four categories: MUST-LEAK, LIKELY-NOT-LEAK, BLOAT, and MAY-LEAK. MUST-LEAK warnings point to true leaks; LIKELY-NOT-LEAK contains warnings that are highly unlikely to be leaks; warnings in BLOAT are also unlikely to be leaks but should be fixed to improve performance; manual validation only needs to be focused on MAY-LEAK warnings which cannot be validated by our tool, leading to significantly reduced human effort and improved productivity. The proposed technique is efficient: it employs (1) path-directed test case generation—the path information in a warning is used to direct concolic testing to generate test cases covering only the reported path; and (2) object-based state tracking (as apposed to pointer-based reference tracking) to efficiently track the status of the allocated memory. Our initial experimental results demonstrate that the technique is promising and can be useful in improving the usefulness of (existing and future) static leak detectors. We plan to incorporate more large, real-world applications in the future studies of the technique. It is also worth investigating whether and how the technique can be extended to validate static analysis warnings for other types of bugs (e.g., buffer overflow) and security vulnerabilities.

ACKNOWLEDGMENTS

Mengcheng Li, Yuanjun Chen, and Linzhang Wang were supported by the National Natural Science Foundation of China (No. 91118007, 61170066, and 61021062) and the National 863 High-Tech Program of China (No. 2012AA011205).

8. REFERENCES

[1] GNU core utilities. http://www.gnu.org/software/coreutils/.
[2] The Klocwork static analysis tool. http://www.klocwork.com/.
[3] US-CERT vulnerability notes database. http://www.kb.cert.org/vuls.
[4] Yices: An SMT solver. http://yices.csl.sri.com/.
[5] The Coverity static analysis tools. http://www.coverity.com/, 2012.
[6] CREST: An automatic test generation tool for C. http://code.google.com/p/crest, 2012.
[7] HP Fortify. https://www.fortify.com, 2012.
[8] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pages 143–162, 2008.
[9] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, pages 12–22, 2011.
[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.
[11] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.
[12] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *OOPSLA*, pages 109–126, 2008.
[13] M. D. Bond and K. S. McKinley. Leak pruning. In *ASPLOS*, pages 277–288, 2009.

[14] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *SCAM*, pages 149–160, 2006.

[15] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI*, pages 480–491, 2007.

[16] J. Clause and A. Orso. LEAKPOINT: pinpointing the causes of memory leaks. In *ICSE*, pages 515–524, 2010.

[17] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.

[18] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS*, pages 329–342, 2013.

[19] W. DePauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.

[20] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, pages 181–192, 2012.

[21] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *SAS*, pages 115–134, 2000.

[22] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994, 2011.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[24] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *FSE*, pages 117–127, 2006.

[25] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *PLDI*, pages 101–111, 2007.

[26] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.

[27] M. J. Harrold and G. Rothermel. Siemens programs. http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[28] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, 1992.

[29] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.

[30] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM*, pages 41–50, 2008.

[31] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.

[32] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *ICSE*, pages 252–261, 2006.

[33] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.

[34] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *MSR*, pages 27–, 2007.

[35] S. Kim and M. D. Ernst. Which warnings should i fix first? In *FSE*, pages 45–54, 2007.

[36] N. Kosmatov. All-paths test generation for programs with internal aliases. In *ISSRE*, pages 147–156, 2008.

[37] T. Kremenek, K. Ashcraft, J. Yang, and D. R. Engler. Correlation exploitation in error ranking. In *FSE*, pages 83–93, 2004.

[38] T. Kremenek and D. Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *SAS*, pages 295–315, 2003.

[39] J. Maebe, M. Ronsse, , and K. D. Bosschere. Precise identification of memory leaks. In *WODA*, pages 25–31, 2004.

[40] B. Meredith. Omega: An instant leak detector tool for valgrind. http://www.brainmurders.eclipse.co.uk/omega.html, December 2008.

[41] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP*, pages 351–377, 2003.

[42] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, pages 65–74, 2007.

[43] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI*, pages 397–407, 2009.

[44] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *SAS*, pages 405–424, 2006.

[45] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, pages 291–302, 2005.

[46] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *ASE*, pages 194–203, 2007.

[47] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *ICSE*, pages 341–350, 2008.

[48] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[49] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA*, pages 254–264, 2012.

[50] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In *ISSTA*, pages 1–11, 2011.

[51] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX*, pages 307–320, 2008.

[52] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *FSE*, pages 115–125, 2005.

[53] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[54] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

[55] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *PLDI*, pages 270–282, 2011.

[56] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE*, pages 151–160, 2008.

[57] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.