LEAKPOINT: Pinpointing the Causes of Memory Leaks

James Clause
College of Computing
Georgia Institute of Technology
clause@cc.gatech.edu

Alessandro Orso
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu

ABSTRACT

Most existing leak detection techniques for C and C++ applications only detect the existence of memory leaks. They do not provide any help for fixing the underlying memory management errors. In this paper, we present a new technique that not only detects leaks, but also points developers to the locations where the underlying errors may be fixed. Our technique tracks pointers to dynamically-allocated areas of memory and, for each memory area, records several pieces of relevant information. This information is used to identify the locations in an execution where memory leaks occur. To investigate our technique's feasibility and usefulness, we developed a prototype tool called LEAKPOINT and used it to perform an empirical evaluation. The results of this evaluation show that LEAKPOINT detects at least as many leaks as existing tools, reports zero false positives, and, most importantly, can be effective at helping developers fix the underlying memory management errors.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Performance, Reliability

Keywords

Leak detection, Dynamic tainting

1. INTRODUCTION

Memory leaks are a type of unintended memory consumption that can adversely impact the performance and correctness of an application. In programs written in languages such as `C` and `C++`, memory is allocated using allocation functions, such as `malloc` and `new`. Allocation functions reserve a currently free area of memory m and return a pointer p that points to m's starting address. Typically, the program stores and then uses p, or another pointer derived from p, to interact with m. When m is no longer needed, the program should pass p to a deallocation function (*e.g.,* `free` or `delete`) to deallocate m. A leak occurs if, due to a memory management error, m is not deallocated at the appropriate time. There are two types of memory leaks: lost memory and forgotten memory. *Lost memory* refers to the situation where m becomes unreachable (*i.e.,* the program overwrites or loses p and all pointers derived from p) without first being deallocated. *Forgotten memory* refers to the situation where m remains reachable but is not deallocated or accessed in the rest of the execution.

Memory leaks are relevant for several reasons. First, they are difficult to detect. Unlike many other types of failures, memory leaks do not immediately produce an easily visible symptom (*e.g.,* a crash or the output of a wrong value); typically, leaks remain unobserved until they consume a large portion of the memory available to a system. Second, leaks have the potential to impact not only the application that leaks memory, but also every other application running on the same system; because the overall amount of memory is limited, as the memory usage of a leaking program increases, less memory is available to other running applications. Consequently, the performance and correctness of every running application can be impacted by a program that leaks memory. Third, leaks are common, even in mature applications. For example, in the first half of 2009, over 100 leaks in the Firefox web-browser were reported [18].

Because of the serious consequences and common occurrence of memory leaks, researchers have created many static and dynamic techniques for detecting them (*e.g.,* [1, 2, 4, 7–14, 16, 17, 20–23, 25, 27, 28]). The adoption of static techniques has been limited by several factors, including the lack of scalable, precise heap modeling. Dynamic techniques are therefore more widely used in practice. In general, dynamic techniques provide one main piece of information: the location in an execution where a leaked area of memory is allocated. This location is supposed to serve as a starting point for investigating the leak. However, in many situations, this information does not provide any insight on where or how to fix the memory management error that causes the leak: the allocation location and the location of the memory management error are typically in completely different parts of the application's code.

To address this limitation of existing approaches, we propose a new memory leak detection technique. Our technique provides the same information as existing techniques but also identifies the locations in an execution where leaks occur. In the case of lost memory, the location is defined as the point in an execution where the last pointer to an unallocated memory area is lost or overwritten. In the case of forgotten memory, the location is defined as the last point in an execution where a pointer to a leaked area of memory was used (*e.g.,* when it is dereferenced to read or write memory, passed as a function argument, returned from a function, or used as an operand in an arithmetic expression). As our evaluation shows,

ICSE '10, May 2-8 2010, Cape Town, South Africa

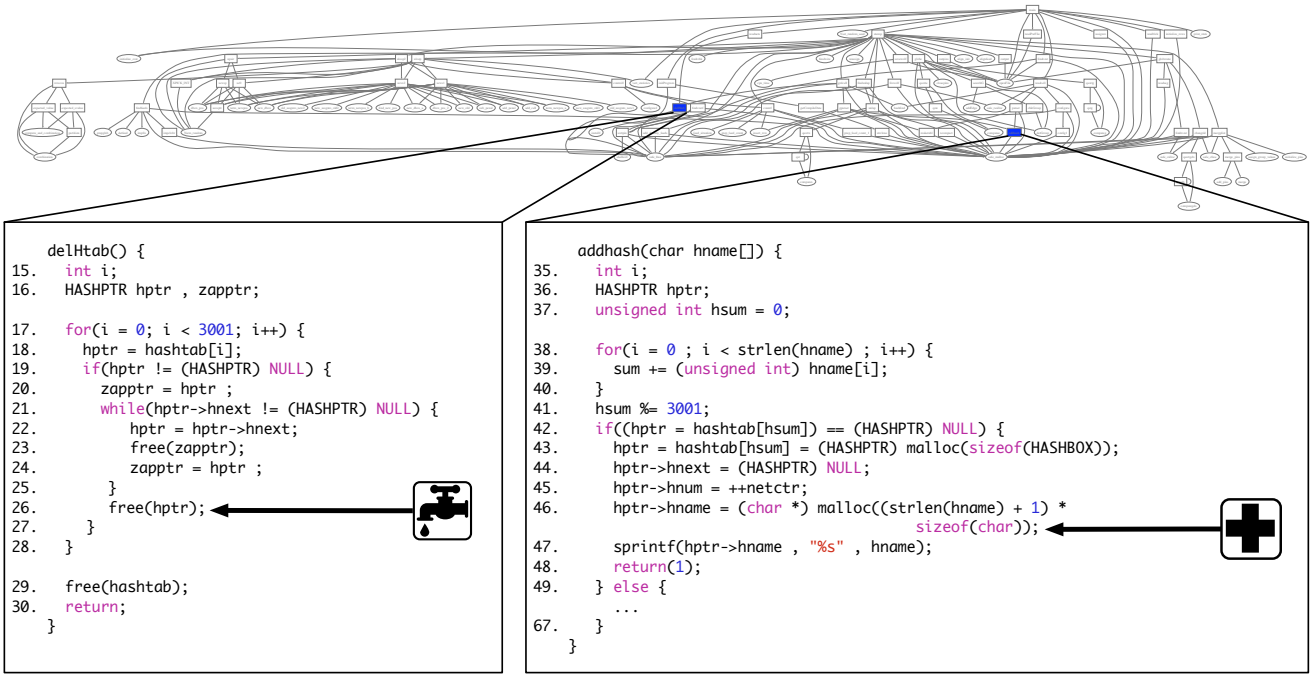

Figure 1: Call graph and relevant code for a memory leak found in `300.twolf`.

identifying the locations where leaks occur can accurately guide developers to the points in the code where the memory management errors that cause the leaks may be fixed.

To evaluate the effectiveness of our technique, we implemented it in a prototype tool called LEAKPOINT and performed an empirical evaluation on a number of subjects. As subjects, we used applications that contain real memory management errors that cause leaks. The results of this evaluation show that, for the subjects that we considered, our technique finds at least as many leaks as existing techniques, reports zero false positives, and can be effective at guiding developers directly to the locations where the memory management errors may be fixed.

The contributions of this paper are:

- A novel technique that identifies the locations in a program's execution where memory leaks occur.
- A prototype tool, LEAKPOINT, that implements our technique.
- An empirical evaluation that shows the effectiveness and usefulness of our technique.

The rest of this paper is organized as follows: Section 2 presents a motivating example for our technique. Section 3 describes our technique in detail. Section 4 presents our empirical evaluation. Sections 5 and 6 discuss related work and present our conclusions.

2. MOTIVATING EXAMPLE

As a motivating example for our technique, we use one of the real leaks that we found in `300.twolf` during our evaluation. `300.twolf` is a computer-aided-design program that calculates the routing and placement of transistors for microchip design [26]. The leak that we are considering occurs when the application is executed using its test-input set, which is provided with the application. Figure 1 shows the dynamic call graph for this execution (top) and relevant portions of the application's code, displayed as call-outs from their respective locations in the call graph (bottom).

The plus icon (⊕) in the right call-out indicates the information that most existing leak detection tools provide for this leak: the location where the leaked memory was allocated. For this example, they would inform a developer that the memory area allocated at line 45 in function `addhash` is leaked during the execution. As we mentioned previously, in most cases, including our example, this information does not provide a developer with any guidance or insight on how to fix the error that causes the leak.

The faucet icon (🚰) in the left call-out indicates the additional information that our technique provides: the location where the leak occurs. For this example, our technique would inform a developer that the area of memory allocated at line 46 in function `addhash` is leaked at line 26 in function `delHtab`. With this additional information, it becomes much simpler to identify (1) the memory management error—for this example, the error is that the memory area pointed to by `hptr` at line 26 in function `delHtab` is deallocated without first deallocating the memory pointed by `hptr->hname`; and (2) the modifications that are necessary for fixing the error—in this case, the necessary modifications consist of inserting an appropriate call to free, "`free(hptr->hname)`" immediately before line 26 in function `delHtab`. In fact, using information provided by our tool (*i.e.,* both the location where the leaked memory area is allocated and the location where it is leaked), we were able to quickly diagnose and fix this error, even though we had never seen `300.twolf`'s code before (see Section 4.4 for details).

This example is a good representative of the type of leak we expect to encounter in practice in real programs. First, it is a real leak that occurs in a released, commonly used application. Second, it is caused by a common programming error: forgetting to deallocate a component of an object before deallocating the object itself. Third, its occurrence does not noticeably impact the application; even though it leaks memory, `300.twolf` runs to completion and produces the correct output. And fourth, the allocation site and the location where the error may be fixed are far apart in the code; they are in two separate functions and, as the call graph shown in Figure 1 illustrates, these two functions do not occur near one another in the call graph.

3. LEAKPOINT TECHNIQUE

In this section, we present our technique for identifying the locations in an execution where leaks occur. We first provide an intuitive description of the technique, and then discuss its main characteristics in detail.

3.1 General Approach

Our technique for leak detection is based on dynamic tainting (or dynamic information flow) [5, 6, 15]. Simply stated, dynamic tainting is an analysis technique that is concerned with tracking information in an application as the application executes. Intuitively, it consists of three parts: (1) *tainting* interesting program data (*e.g.,* variables or memory locations) with a taint mark, (2) *propagating* taint marks, along with their associated data, as the program executes, and (3) *checking* which taint marks are associated with program data at specific points in the execution.

Specifically, our technique uses dynamic tainting to track pointers to dynamically-allocated areas of memory as a program executes. During an execution, when an area of memory m is allocated, the technique creates a new taint mark t_m and uses t_m to taint the returned pointer. The technique stores five additional pieces of information with each taint mark. (Collectively, we refer to these pieces of information as taint mark metadata.)

- *Allocation location*: the location in the execution where memory area m is allocated.[1]
- *Allocation size*: the size in bytes of memory area m.
- *Deallocated indicator*: a boolean flag that indicates whether memory area m has been deallocated.
- *Pointer count*: the number of pointers that are tainted with t_m (*i.e.,* the number of pointers that currently point to m).
- *Last use location*: the location in the code where a pointer tainted with t_m was last used (*i.e.,* the location where it was last dereferenced to access m, passed as a function argument or return value, manipulated using an arithmetic or bitwise operator, or copied using an assignment operator).[1]

Note that the allocation location and allocation size are immutable, whereas the deallocation indicator, pointer count, and last use location are mutable; they are updated by the technique as it propagates taint marks (see Section 3.3).

While propagating taint marks and updating taint mark metadata during an execution, the technique performs two checks. Each check is designed to detect one type of leak. To detect lost memory, each time a pointer tainted with a taint mark t_m is lost or overwritten, the technique checks whether that pointer was the last pointer to m. If it was, the technique reports the current location in the execution as the location where m is leaked. To detect forgotten memory, at the end of an execution, the technique checks which memory areas have not been deallocated. For each area of memory m that was not deallocated, the technique reports the corresponding taint mark t_m's last use location as the location where m is leaked. In rest of this section, we describe in detail the three parts of our technique: tainting, propagating, and checking.

[1]The current implementation of our technique uses stack traces with line number information to represent location information.

3.2 Tainting

This part of our technique is responsible for assigning taint marks to pointers to dynamically-allocated memory areas. To taint such pointers, our technique intercepts all calls to allocation functions. By default, the technique intercepts calls to system provided allocation functions (*i.e.,* `malloc`, `calloc`, `realloc`, `new`, and `new[]`). However, for some applications, only considering these functions limits the technique's ability to detect leaks. For example, this is the case for applications that use memory pools—a common approach for avoiding the overhead of dynamic memory allocation. At a high level, a memory pool is a single, large dynamically-allocated memory area that is divided into smaller memory chunks that are managed using custom allocation and deallocation functions. Without considering these custom functions, our technique can only check whether the entire memory pool is leaked; it cannot check whether any of the chunks of manually-managed memory are leaked during an execution.

To support leak detection inside memory pools and handle other similar situations, our technique allows developers to indicate which functions in a program should also be considered allocation and deallocation functions. Additional allocation functions must fulfill the same requirements as the system provided allocation functions; they must (1) take as input the size of the memory area to allocate and (2) return either a pointer to the allocated area of memory or `NULL` if the allocation is unsuccessful.

Regardless of whether an allocation function is in the default set or added by a programmer, the technique operates in the same way. When an allocation function successfully allocates an area of memory m (*i.e.,* the return value is not `NULL`), the technique creates a new taint mark t_m and initializes t_m's taint mark metadata: the allocation location and last use location are initialized to the current location in the execution; the allocation size is initialized to the size passed as a parameter to the allocation function; the deallocation indicator is set to false; and the pointer count is initialized to one to indicate that there is currently a single pointer to m. Then, the technique taints the pointer that is returned by the allocation function with t_m.

3.3 Propagating

This part of our technique performs two main tasks: tracking pointers throughout an execution and updating taint mark metadata. We describe each of these tasks in the following subsections.

3.3.1 Tracking Pointers

The first task of the propagation part is to track the flow of pointers through an execution by propagating taint marks. In order to correctly propagate taint marks, our technique must use a propagation policy that correctly models all operations that can be performed on pointers. In many situations, creating such a model is challenging because of the ambiguity between pointers and integers; depending on the context in which an operator is used, the result of an operation may be either a pointer or an integer. The propagation policy that our technique uses is the result of combining information about the semantics of C and C++ and knowledge of (and intuition about) the common usage of operators. We present our technique's propagation policy by discussing how it models each operator.

Arithmetic operators.

Assignment ($c = a$): In the case of an assignment, the left-hand side is simply tainted with the same taint mark that is associated

with the right-hand side. If a is tainted, c is tainted with the same taint mark, whereas if a is not tainted, c is not tainted.

Addition ($c = a + b$): In the case of addition, we consider four different cases depending on the taint marks associated with the operands that are being added.

c	a	b
t_m	t_m	–
t_m	–	t_m
–	t_m	t_n
–	–	–

Each of the four cases is shown in the above table. The first and second rows illustrate how the propagation policy models situations where a numeric offset (not tainted) is added to a pointer to an area of memory m (tainted with t_m). In this situation, the most likely result of the addition is a pointer that points inside memory area m (*e.g.,* when an offset is added to pointer to index into a dynamically-allocated array). Therefore, the result is tainted with t_m. The third and fourth rows illustrate how the propagation policy models situations where two pointers (tainted with t_m and t_n, respectively) or two non-pointers (not tainted) are added. In both of these situations, it is unlikely that the result of the addition is a pointer and the result is therefore not tainted.

Subtraction ($c = a - b$): Subtraction is handled in a manner similar to addition and also involves four cases.

c	a	b
t_m	t_m	–
–	–	t_n
–	t_m	t_n
–	–	–

The first row in the above table illustrates how the propagation policy models situations where a numeric offset (not tainted) is subtracted from a pointer to an area of memory m (tainted with t_m). Like for addition, in this situation the most likely result of the subtraction is a pointer that points inside memory area m. Therefore, the result is tainted with t_m. Unlike addition, however, subtraction is not commutative; a pointer subtracted from a numeric offset is most likely not a pointer. This situation is illustrated in the second row in the table; the result of subtracting a tainted operand from a non-tainted operand is not tainted. The remaining two cases, shown in the third and fourth rows, are again similar to addition. The result of subtracting two pointers or two non-pointers is most likely not a pointer and therefore is not tainted.

Multiplication, division, modulus: We could not identify any situation where the result of these operators should be considered a pointer. Therefore, regardless of the taint marks associated with the operands, the result of these operations is never tainted.

Bitwise operators.

And ($c = a \ \& \ b$): Bitwise and is also handled similarly to addition and involves the same four cases.

c	a	b
t_m	t_m	–
t_m	–	t_m
–	t_m	t_n
–	–	–

The first and second rows in the above table illustrate how the propagation policy handles situations where a numeric offset (not tainted) is anded with a pointer (tainted with t_m). Although this situation is not common, it does occur in certain cases, such as when byte-aligning dynamically-allocated memory (*i.e.,* ensuring that the starting address of a memory area is a multiple of a given size). This case is conceptually similar to indexing into an array by adding an offset to a pointer; the result is a pointer to the same area of memory that the pointer operand points to. Therefore, in this situation, the result is tainted with t_m. The third and fourth rows illustrate how the propagation policy models situations where two pointers or two non-pointers are anded together; again, the result is most likely not a pointer and therefore is not tainted.

Or, xor, shift, not: Similar to multiplication, division, and modulus, we could not identify any situation where the result of these operators should be considered a pointer. Therefore, regardless of the taint marks associated with operands, the result of these operations is never tainted.

Comparison operators

Less than, greater than, less than or equal to, greater than or equal to, equal to, not equal to, and, or, not: The result of these operators is never a pointer and therefore is never tainted.

As we mentioned previously, accurately modeling operators that operate on pointers is challenging, as it is possible for developers to use any operator in creative and inventive ways. Therefore, it is unlikely that a propagation policy can be proven to be sound and complete. As far as our policy is concerned, as stated previously, we defined it based on domain knowledge and on experience, and we verified that it works correctly for all of the software that we studied so far, as discussed in Section 4. If additional experimentation would reveal shortcomings of our propagation policy, we will refine it accordingly.

3.3.2 Updating information

The second task of the propagation part of our technique is to update the mutable pieces of taint mark metadata: pointer counts, deallocation indicators, and last use locations. We describe how our technique updates each piece of information separately.

To correctly update pointer counts, the technique must handle assignments, function returns, and deallocations of dynamically-allocated memory areas.

Assignment operator: To handle the assignment operator, the technique performs two actions in addition to propagating taint marks. For an assignment statement, $c = a$, the technique first checks whether c is currently tainted. If c is tainted with a taint mark t_m, the technique decrements t_m's pointer count because a pointer to memory area m is being overwritten. Then, as previously described, the technique assigns the taint mark associated with a (if any) to c. Finally, it checks whether c is tainted after the propagation. If c is tainted with taint mark t_n, the technique increments t_n's pointer count because a new pointer to memory area n was created as a result of the assignment.

Function return: Our technique intercepts all function return events. For each of the returning function's local variables, the technique checks whether the variable is a pointer (*i.e.,* whether it is tainted). If it is, the technique decrements the pointer count stored in the taint mark associated with the local variable. The pointer count is decremented because, after the function returns, the local variable is no longer in scope and cannot be accessed by the program.

Deallocation of dynamically-allocated memory: Dynamically-allocated memory is deallocated using an explicit call to a deallocation function (*i.e.,* `free`, `delete`, `delete[]`), which takes as a parameter a pointer to the area of memory that should be deallocated. (Programmer-defined deallocation functions are supported in the same manner as programmer-defined allocation functions—

see Section 3.2.) The technique intercepts deallocation functions and identifies the taint mark t_m that is associated with the pointer passed to the deallocation function. It then searches memory area m, checking whether m contains any pointers. For each pointer p that it finds, the technique decrements the pointer count of the taint mark associated with p. This handles situations where memory is indirectly leaked (*i.e.,* the only pointer to memory area n is stored inside memory area m and, when m is deallocated, n is leaked).

Updating deallocation indicators and last use locations is simpler than updating the pointer counts. To update deallocation indicators, the technique again intercepts deallocation functions, identifies the taint mark associated with the pointer that is passed to the function, and sets the taint mark's deallocation indicator to true. To update last use locations, the technique sets the last use location of a taint mark t_m to the current location in the execution each time a pointer tainted with t_m is (1) propagated, as described in Section 3.3.1, (2) passed as a function argument, (3) returned from a function, or (4) used to access memory area m.

3.4 Checking

This third and last part of our technique, checking, is also responsible for two tasks: identifying when leaks occur and generating the leak reports presented to developers. Leak reports comprise three pieces of information: the location where the leaked area of memory was allocated, the location where the leak occurred, and the size of the leaked memory area. Presenting the sizes of leaked memory areas does not necessary help developers identify the locations where the memory management errors may be fixed, but it does provide a mechanism for judging the relative severity of a particular memory leak. Typically, leaks of larger areas of memory should be investigated before leaks of smaller areas.

To identify when memory leaks occur, our technique performs two checks; one to detect lost memory and the other to detect forgotten memory. To detect lost memory, the technique uses pointer counts. Each time the pointer count of a taint mark t_m is decremented, the technique checks whether the count's value after the decrement is zero. If the count is zero, and t_m's deallocation indicator is false, the technique classifies m as lost memory and generates a lost memory leak report. The report's allocation location and size information are initialized to the allocation location and the allocation size stored in taint mark t_m, respectively. The leak location is initialized to the current location in the execution. After generating the report the technique deallocates memory area m. This causes the technique's deallocation of dynamically-allocated memory handler, described in Section 3.3.2, to update the pointer count of any taint marks associated with pointers stored inside m. The recursive interaction of these two parts of the technique allows it to detect indirect leaks that are caused by the leaking of m.

The check used to detect forgotten memory is performed at the end of an execution. The technique first identifies all taint marks whose deallocation indicator is set to false; because the deallocation indicator gets set when memory areas are deallocated, each taint mark with a false deallocation indicator corresponds to an area of memory that has not been deallocated. The technique classifies each such area of memory m as forgotten memory and generates a forgotten memory leak report. Like for lost memory leak reports, the allocation location and size for forgotten memory leak reports are initialized to the corresponding taint mark t_m's allocation location and allocation size. However, unlike for lost memory leak reports, the leak location is set to t_m's last use location.

Because programs often repeat sequences of calculations, it is possible for multiple leak reports to have both the same allocation location and the same leak location. Therefore, to avoid overwhelming developers with identical leak reports, reports that have the same allocation location and the same leak location are merged. The allocation location and leak location of the merged report are the same as its component reports, while the merged report's size is the sum of the sizes in the component reports. Furthermore, our technique sorts leak reports by two criteria: type (lost memory leak reports are presented before forgotten memory leak reports) and size (leak reports for larger areas of memory are presented before leak reports for smaller areas of memory).

It is worth noting that, although our technique is currently defined to generate leak reports when memory leaks are detected, it can be easily modified to support other actions. For example, when detecting a leak, the technique could attach a debugger to the running program or it could terminate the execution. It is also possible to support more complex actions, such as only generating leak reports after a specified amount of memory is leaked. The specific action chosen may depend on the context or on the goals of the developers who are using the technique.

4. EMPIRICAL EVALUATION

To assess the effectiveness, and usefulness of our technique we implemented it in a prototype tool, called LEAKPOINT, and investigated the following research questions:

RQ1: How does LEAKPOINT's ability to detect memory leaks compare to existing leak detection tools?

RQ2: How effective is LEAKPOINT at guiding developers to the locations where the memory management errors that cause memory leaks may be fixed?

Note that, although by definition our technique should be able to detect all leaks, RQ1 is a relevant research question: the difficulties of tracking pointers that we describe in Section 3.3.1 may cause LEAKPOINT to miss some leaks or report spurious ones. Comparing LEAKPOINT against existing tools can provide a better understanding of how our technique will perform in practice and provides evidence that the technique's propagation policy is accurate.

In the following subsections we discuss LEAKPOINT, our subjects, and our experimental protocol and results for each research question. We also present a small case study in which we investigated the performance of LEAKPOINT in terms of the runtime overhead that it imposes on running applications.

4.1 Prototype tool

LEAKPOINT is a prototype implementation of our technique for Linux/x86 binaries that is built as a Valgrind tool. Valgrind is a generic binary instrumenter that is optimized to support heavyweight dynamic analyses [19]. We chose to use Valgrind because it operates at the binary level, which allows our technique to easily handle shared libraries. Source-level techniques would require recompilation of every library an application may use, a task that may be difficult or even impossible (*i.e.,* if source code is unavailable). Moreover, Valgrind abstracts away much of the complexity of the underlying operating system and architecture. This means that LEAKPOINT can easily support multiple operating system and architecture combinations (*i.e.,* Linux on x86, amd64, ppc32, and ppc64 and Darwin (Mac OS X) on x86) with limited additional implementation effort.

We leverage Valgrind's built-in functionality to implement the three parts of our technique. To taint pointers to dynamically-allocated areas of memory, we use Valgrind's function-interception capabilities, which allow us to intercept all calls to allocation and deallocation functions. To implement taint mark propagation and

Subject	Description	LoC	# Detected memory leaks (# false positives)							
			mtrace	*omega*	*MemCheck*			LEAKPOINT		
					lost	forgotten	total	lost	forgotten	total
164.gzip	Compression	5,606	4	1	1	3	4	1	3	4
175.vpr	FPGA circuit placement	11,316	47	0	0	47	47	0	47	47
176.gcc	Compiler	129,907	1121	406 (1415)	255	869	1121	255	869	1121
181.mcf	Vehicle scheduling	1,482	0	0	0	0	0	0	0	0
186.crafty	Chess	12,907	37	0	0	37	37	0	37	37
197.parser	Word processing	7,763	2	0	0	2	2	0	2	2
252.eon	Computer visualization	22,265	380	380	380	0	380	380	0	380
253.perlbmk	Programming language	69,460	536	0 (2)	0	3481	3481	0	3481	3481
254.gap	Group theory	35,698	2	0	0	2	2	0	2	2
255.vortex	Database	49,226	15	1	1	14	15	1	14	15
256.bzip2	Compression	3,236	10	1	1	9	10	1	9	10
300.twolf	Computer aided design	17,849	1403	68 (3)	68	1335	1403	68	1335	1403

Table 1: Applications in the first set of subjects and results for RQ1.

Subject	Description	LoC	# Errors
gcc 3.0	Compiler	353,620	1
lighttpd 1.4.19	Web server	51,163	2
transmission 1.20	Bittorrent client	82,351	1

Table 2: Applications in the second set of subjects.

checking, we created a set of functions, one for each instruction in Valgrind's intermediate representation. These functions model the semantics of the instructions and implement the appropriate propagation and checking functionality. We use Valgrind's instrumentation capabilities to insert a call to the appropriate function before each instruction. Finally, we developed a client API for supporting custom memory management functions, described in Section 3.2. To use this API, developers insert calls in their code that inform LEAKPOINT when memory areas are allocated and deallocated by their custom routines. However, to remove a potential source of bias, we did not use any of these functions in our evaluation.

4.2 Subjects

In our empirical evaluation, we used two sets of subjects. The *first set* consists of the twelve applications from the integer portion of the SPEC CPU2000 benchmark suite version 1.0 [26]. Table 1 lists these applications and shows, for each application, a brief description of the application's functionality (*Description*) and size in lines of code (*LoC*). We chose these applications for several reasons. First, they cover a wide range of problem domains and range in size from ≈1.5k LoC (`181.mcf`) to ≈129k LoC (`176.gcc`). Second, they contain numerous memory management errors, which means that they are good subjects for investigating RQ1. And third, the SPEC benchmarks were designed to evaluate performance, so the applications are also appropriate subjects for investigating the runtime overhead imposed by LEAKPOINT. The SPEC benchmarks are also distributed with several sets of inputs, which we used when running the applications.

The *second set* consists of applications that contain documented memory management errors that were fixed by the application's original developers. Because we need documentation of how the leaks were fixed, we were unable to reuse subjects from related work. The subjects that they used were either proprietary (*e.g.,* [9]) or did not have a record of how the leaks were fixed (*e.g.,* [14]). To select subjects that meet this requirement, we surveyed on-line bug databases and issue trackers for large, commonly-used applications. Although leaks are common, they are often difficult and time consuming to reproduce based solely on information provided in bug reports. In fact, none of the bug reports we encountered in our survey provided correct, detailed steps on how to reproduce the leak; at best, they provided a general description of the necessary conditions for the leak to occur. As subjects, we chose the applications which contain the first four leaks that we were able to reproduce successfully: `gcc` version 3.0, which contains one memory management error (note that this version is different from the one that is included in the SPEC benchmarks),[2] `lighttpd` version 1.4.19, which contains two memory management errors,[3,4] and `transmission` version 1.20, which contains one memory management error.[5] Information about these applications is shown in Table 2. This second set of applications is ideal for investigating RQ2. The fact that the memory management errors were fixed by the original developers means that we can assess, in an unbiased way, how effective LEAKPOINT is at identifying the locations where the memory management errors may be fixed.

4.3 RQ1

To gather the necessary data for investigating RQ1, we used the twelve applications from our first set of subjects. We ran each application using its test-input set and checked them for leaks using LEAKPOINT and three other leak detection tools: `mtrace` [1], `MemCheck` [25], and `omega` [16]. We chose these tools because they are freely available and are widely used. `MemCheck` and `mtrace` only detect memory leaks, while `omega` provides information that is similar to what is provided by our technique. (Additional information about the techniques these tools use for detecting leaks is presented in Section 5.) For each leak report generated by each tool, we manually verified whether it corresponds to an actual leak or is a false positive.

Table 1 shows, for each subject, the number of leak reports generated by each tool which correspond to actual leaks and, in parentheses, the number of leak reports that are false positives. To eliminate clutter in the table, false positive counts of zero are not shown. For LEAKPOINT and `MemCheck`, the table also shows how many of the leak reports are for lost memory and how many are for forgotten memory. We are unable to provide this information for `mtrace` or `omega` because `mtrace` does not distinguish between the two types of leaks and `omega` only detects lost memory. We are also unable to determine if there are any false negatives because the total number of leaks in a program is generally unknown.

As the table shows, LEAKPOINT, `MemCheck`, and `mtrace` report zero false positives and detect an identical number of leaks

[2] `http://gcc.gnu.org/ml/gcc-cvs/2004-06/msg01124.html`
[3] `http://redmine.lighttpd.net/issues/show/1774`
[4] `http://redmine.lighttpd.net/issues/show/1775`
[5] `http://trac.transmissionbt.com/ticket/961`

for all subjects except 253.perlbmk, for which mtrace reports fewer leaks. This difference is due to the fact that mtrace fails to record a large number of calls to malloc that allocate memory that is subsequently leaked. These calls are not intercepted because mtrace uses malloc hooks, which are known to be unreliable. (In fact, malloc hooks, and presumably mtrace, will be removed from future versions of the GNU C library.)[6]

The result that LEAKPOINT and MemCheck have the same performance is not unexpected; MemCheck is a widely used (and thus extensively tested) tool, and it would be surprising if LEAKPOINT were able to detect more leaks than MemCheck can detect. This is an important result nevertheless, as it provides strong evidence that our technique is as effective as existing tools in identifying memory leaks, while potentially providing more beneficial information for developers. In addition, the result is a sanity check on the correct implementation of LEAKPOINT. The table also shows that LEAKPOINT outperforms omega in several ways. First, as we mentioned previously, omega is unable to detect forgotten memory. As the table shows, forgotten memory is common, and not handling it may cause a large number of leaks to go undetected. Second, unlike LEAKPOINT, omega generates false positives for three of the subjects. These drawbacks reduce omega's effectiveness and limit its usefulness in practice.

Like any empirical evaluation, our investigation of RQ1 is limited in scope, and its results may not generalize. However, our subjects are real programs that span a wide range of application types and contain actual leaks. Therefore, although further evaluation is needed, we believe that our results are promising. They show that LEAKPOINT is likely to be (1) as effective at detecting leaks as tools that focus only on leak detection and (2) more effective than existing tools that provide the same type of information as our technique.

4.4 RQ2

To gather the data necessary for investigating RQ2, we ran each application from our second set of subjects while checking them for leaks using LEAKPOINT. As an initial metric for judging the effectiveness of the technique when helping developers fix memory management errors, we compared the location of the leak reported by LEAKPOINT against the location where the memory management error was fixed. In future work, we plan on performing user-studies to better assess the performance of our technique in a fully realistic setting. The remainder of this section presents a detailed discussion of the technique's performance when using our initial metric for each of the considered errors.

Figure 2a shows the relevant portion of code for the error in gcc. Memory allocated at line 4540 in function push_string is leaked when gcc's type verifier switches from an inner-context to an outer-context. The leak occurs at line 5187, where spelling_base is overwritten but the memory area it points to is not deallocated. The commented code in the figure shows the code that was added by the developers to fix this error: a call to free was added at line 5179 to deallocate the memory area before spelling_base is overwritten. LEAKPOINT identifies line 5187 as the location where the leak occurs. This location is close to the location where the memory management error was fixed by the developers; it is in the same function and less than 10 lines away. Moreover, the error could indeed be fixed by adding the call to free directly before line 5187. The reason why the developers chose to place the call at line 5179 instead is most likely because lines 5180–5187 are conceptually an atomic region of code—together, they restore values stored inside the structure pointed to by p.

The first error in lighttpd causes a memory leak if the option url.rewrite-repeat is set in the web server's configuration file. Figure 2b shows the relevant portion of lighttpd's code. Memory allocated at line 429 in mod_rewrite_uri_handler is leaked if this section of code is executed twice. In the first execution, the only pointer to the allocated memory area is stored in the plugin context array at line 430. During the second execution, this pointer is overwritten and, because the area of memory it points to is not deallocated, a leak occurs. As in Figure 2a, the commented lines in Figure 2b show the code that was added by the developers to fix this memory management error; the code now checks whether memory was already allocated. LEAKPOINT identifies line 430 as the location where the leak occurs. This location is even closer to the location where the memory management error was fixed than in the previous case; the two locations actually overlap.

The second error in lighttpd causes a leak when the web server parses a request with duplicated http headers. Figure 2c shows the relevant portion of lighttpd's code for discussing this error. Memory allocated at line 775 in http_request_parse is leaked because the function returns without deallocating it. To fix this error, the developers inserted the commented code at line 826. Because lighttpd is concerned with performance, it maintains a list of allocated request headers that it reuses to save the overhead of memory allocation. The inserted call fixes the error by adding the allocated memory area to the pool of request headers. LEAKPOINT identifies line 825 as the location where the leak occurs. Like for the first leak in lighttpd, LEAKPOINT precisely identifies the location where the memory management error was fixed; the location of the fix is immediately before the location identified by LEAKPOINT.

Figure 2d shows the relevant portion of code for the error in transmission. This error causes memory allocated at line 718 in invokeRequest to be leaked when the download of the corresponding torrent file is stopped. The developers fixed this error by inserting a call to a deallocation function at line 295 in onStoppedResponse, which is called at line 77 in processCompletedTasks. LEAKPOINT identifies line 82 in processCompleteTasks as the location where the leak occurs. Also in this case, the location identified by LEAKPOINT is near the location where the memory management error was fixed. Although the locations are in separate functions, they are executed in close proximity to each other—only 6 statements apart. Moreover, we verified that the memory management error can also be fixed by moving the call to the deallocation function immediately before line 82 in processCompletedTasks.

To gather further evidence of the technique's effectiveness, we also examined the leak reports generated by LEAKPOINT for RQ1. Because we do not have a developer-provided fix for these leaks, we cannot perform the same evaluation that we performed for the four leaks described above. Instead, for each detected leak, we investigated whether the reported location was a suitable point for fixing the leak by (1) inserting a memory deallocation statement at the location indicated by LEAKPOINT and (2) rerunning the application to verify that the leak no longer occurs. The results for these additional leaks, although anecdotal in nature, were encouraging; all of the locations reported by LEAKPOINT proved to be appropriate locations for introducing fixes to the memory errors (*i.e.,* adding a deallocation statement at the reported location prevented the leak from occurring).

Based on these results, we can make some initial observations about the effectiveness of our technique in helping developers un-

[6] http://sources.redhat.com/ml/libc-alpha/2006-07/msg00108.html

```
static struct spelling *spelling_base;

static void push_string(char *string) {
  ...
4540.  spelling_base =
         xmalloc(spelling_size *
                 sizeof(struct spelling));
  ...
}

void finish_init() {
  ...
5179.  // free(spelling_base);
5180.  constructor_decl = p->decl;
  ...
5187.  spelling_base = p->spelling_base;
  ...
}
```

(a) **Relevant code for the error in `gcc`.**

```
URIHANDLER_FUNC(mod_rewrite_uri_handler) {
...
428.  // if (con->plugin_ctx[p->id] == NULL) {
429.       hctx = handler_ctx_init();

430.       con->plugin_ctx[p->id] = hctx;
431.  // } else {
432.  //   hctx = con->plugin_ctx[p->id];
433.  // }
  ...
}
```

(b) **Relevant code for the first error in `lighttpd`.**

```
int http_request_parse(server *srv,
                       connection *con) {
 ...
774.  if (NULL == (ds =
        (data_string *)array_get_unused_element(
             con->request.headers, TYPE_STRING))) {

775.    ds = data_string_init();
 }
 ...
812.  else if (cmp > 0 && 0 == (cmp =
        buffer_caseless_compare(CONST_BUF_LEN(ds->key),
        CONST_STR_LEN("Content-Length")))) {

814.    char *err
815.    unsigned long int r;
816.    size_t j
817.    if (con_length_set) {
818.      con->http_status = 400;
819.      con->keep_alive = 0;
820.      if (srv->srvconf.log_request_header_on_error) {
821.        log_error_write(srv, __FILE__, __LINE__, "s",
                           "duplicate ...");
822.        log_error_write(srv, __FILE__, __LINE__,
                          "Sb", "request-header:\n",
                          con->request.request);
823.      }
824.      // array_insert_unique(con->request.headers,
                               (data_unset *)ds);

825.      return 0;
 }
 ...
}
```

(c) **Relevant code for the second error in `lighttpd`.**

```
static void invokeRequest(void * vreq) {
  ...
718.  hash = tr_new0(uint8_t,
                   SHA_DIGEST_LENGTH);
719.  memcpy(hash, req->torrent_hash,
              SHA_DIGEST_LENGTH);
720.  tr_webRun(req->session, req->url,
                req->done_func, hash);

721.  freeRequest(req);
}

void tr_webRun(tr_session *session,
               ...
               void *done_func_user_data) {
169.  struct tr_web_task * task;
  ...
174.  task->done_func_user_data = done_func_user_data;
  ...
177.  tr_runInEventThread(session, addTask, task);
  ...
}

static void processCompletedTasks(tr_web *web) {
  ...
 77.  task->done_func(web->session,
                     ...
                     task->done_func_user_data);
  ...
 80.  evbuffer_free(task->response);
 81.  tr_free(task->url);
 82.  tr_free(task);
  ...
}

static void onStoppedResponse(tr_session *session,
                              ...
                              void *torrent_hash) {
294.  dbgmsg(NULL, "got a response ... message");
295.  // tr_free(torrent_hash);
296.  onReqDone(session);
}
```

(d) **Relevant code for the error in `transmission`.**

Figure 2: Excerpts of code that illustrate our technique's ability to guide developers to the locations where memory leaks may be fixed.

derstand and eliminate leaks. For the four leaks with developer provided solutions, the location of each leak identified by LEAKPOINT was close to, if not coinciding with, the location where the memory management error was fixed. And for the leak reports without developer provided solutions, adding deallocation statements at the identified locations fixed the considered leaks. These results, albeit preliminary in nature, strongly suggest that our technique can be effective in guiding developers to the locations where the memory management errors that cause leaks may be fixed.

4.5 Runtime overhead

To investigate the runtime overhead that LEAKPOINT imposes, we used the twelve applications from our first set of subjects. We ran each application using its reference-input set twice, once normally and once while checking for leaks using LEAKPOINT, and compared the execution times of these runs. Based on these measurements, we found that LEAKPOINT imposes a runtime overhead of 100–300 times. Although this is a considerable overhead, it is comparable to the 100–200 times reported by tools that provide the same type of information as our approach [14, 16]. Identifying the locations of leaks is inherently expensive because it requires adding instrumentation for nearly every instruction. However, we believe that the detailed information provided by our technique justifies its cost. In our experience, developers will accept high overheads for tools that produce accurate results. This is especially true when, as is the case for LEAKPOINT, the tools do not require any developer interaction and can be run overnight, possibly as part of an automated build system whose results are inspected by developers the next day. In addition, there are several possibilities for reducing LEAKPOINT's overhead that we plan on investigating.

First, LEAKPOINT is an unoptimized prototype. It may be possible to reduce its overhead by applying some recently described optimizations for tainting-based approaches [3, 24]. Second, it is also possible to reduce LEAKPOINT's overhead by using a two-phase approach. In the first phase, the technique could perform a lightweight leak detection that simply identifies the location of memory allocations that are leaked (like most existing approaches do). In the second phase, it could use the fully-fledged approach to monitor only such allocations, rather than every allocation in the program. Such an approach could potentially decrease the overhead imposed by the technique dramatically, while still providing developers with information that guides them to the locations where the memory management errors may be fixed.

5. RELATED WORK

As we mentioned in the introduction, the common occurrence and serious consequences of memory leaks have resulted in a large body of research describing techniques for detecting them. In this section, due to space considerations, we limit our discussion to the most closely related dynamic techniques for C / C++ applications.

Valgrind's `Memcheck` tool [25], Mac OS X's `leaks` tool [13], and `purify` [8] operate at the binary level and use an approach inspired by mark-and-sweep garbage collection. During a program's execution, these tools track memory allocation and deallocation and record the starting address and size of every allocated block of memory. To detect leaks, they scan the application's heap looking for pointers to allocated memory blocks. The scan classifies blocks into three categories: non-leaked (pointers to the start of the block exist), possibly leaked (only pointers to the interior of the block exist), and leaked (no pointers to the block exist). Unlike our technique, these tools do not provide any assistance in identifying the location where the underlying memory management errors could be fixed.

The GNU C library provides a facility for debugging memory allocations that is commonly known as `mtrace` [1]. This approach intercepts calls to `malloc`, `realloc`, and `free` and logs allocation and deallocation events. The logged events are then post-processed to match allocations with deallocations and reveal any allocations without a corresponding deallocation. Because `mtrace` only observes allocations and deallocations, it is also incapable of providing any information beyond the location where the leaked memory is allocated.

Parasoft's `Insure++` tool uses a source-level technique that identifies the locations in an application where pointers are lost or overwritten due to an assignment statement or function return [22]. At runtime, each time one of these locations is executed, the tool uses the same mark-and-sweep approach that `memcheck`, `leaks`, and `purify` use to identify leaks. This approach allows `Insure++` to provide information that is similar to what our technique provides. However, like `omega` it cannot detect forgotten memory. Also, because `Insure++` does not handle shared libraries, it is unable to identify memory leaks that originate or occur inside them (*i.e.,* when memory is allocated or leaked within a shared library). We were unable to directly compare LEAKPOINT against `Insure++` because Parasoft did not respond to our request for an evaluation copy of `Insure++`.

The technique proposed by Maebe and colleagues [14] and independently implemented by Meredith in the `omega` tool [16] (which we compared against LEAKPOINT in Section 4.3) is most similar to our approach. Their technique also maintains a pointer count for each area of allocated memory, but instead of tracking the flow of pointers, it intercepts all writes to memory. Each time a value v is written to memory area m, v and the content of m are checked to see whether they are pointers. A value is considered to be a pointer if it is equal to the starting address of one of the currently allocated areas of memory. If the contents of m are considered to be a pointer, the pointer's count is decremented and a leak report is generated if the count is zero. If v is considered to be a pointer, then its count is incremented. Unlike LEAKPOINT, which accurately tracks pointers using its propagation policy, Maebe and colleagues' technique uses a heuristic approach to identify pointers. Consequently, their technique can generate both false positives, as demonstrated in Section 4.3, and false negatives [14]. Moreover, the combination of using a heuristic and only considering memory writes can also lead to inaccuracies in identifying the locations where leaks occur. For example, when run on the memory management error in `gcc` presented in Section 4.4, `omega` identifies line 6998 in function `cse_insn` as the location where the error should be fixed. This location is very far from the location where the actual fix was made; they are in separate functions and different files. Most importantly, we were unable to prevent this leak by adding a deallocation statement at the identified location. Finally, as we mentioned in Section 4.3, Maebe and colleagues' technique does not detect forgotten memory.

Hauswirth and Chilimbi developed a statistical leak detection technique called `SWAT` [9]. `SWAT` is similar to Maebe and colleagues' technique in that it examines memory accesses and uses a heuristic to identify pointers. However, unlike their technique, `SWAT` samples accesses and uses a user-provided timeout value to detect leaks; if an area of memory is currently allocated and has not been touched by an observed memory access within the length of the timeout, a leak is reported. Sampling greatly reduces the overhead of the technique and allows it to be used on live applications, which is a significant benefit. However, sampling also increases the number of false positives generated by the approach. When run on the applications in our first set of studies, Hauswirth and Chilimbi

report that the false positive rate, depending on the sampling rate, ranges from 3 to 35%. In comparison, LEAKPOINT reports zero false positives for these subjects. Sampling can also exacerbate the inaccuracies caused by using a heuristic and by only considering memory accesses when identifying the locations where leaks occur (*i.e.,* memory accesses that are not observed will not update the accessed memory's last use location). We could not directly compare LEAKPOINT against SWAT because SWAT is not publicly available.

6. CONCLUSIONS

In this paper we presented a novel leak detection technique that not only detects leaks, but also identifies the locations in an execution where leaks occur. Our approach uses taint marks to identify and maintain information about pointers to dynamically-allocated areas of memory. Taint marks are propagated as an application executes and are checked to identify the locations where leaks occur. The identified locations are then presented to developers as locations where the memory management errors that cause the detected leaks may be fixed.

We also presented LEAKPOINT, a prototype tool that implements our technique. In the evaluation of our technique, we used LEAKPOINT to detect memory leaks in programs from the SPEC benchmarks as well as several real-world applications. The results of the evaluation show that, for the subjects that we considered, LEAKPOINT detects at least as many memory leaks as existing tools, reports zero false positives, and can be effective at helping developers understand and fix memory errors.

Acknowledgments

This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

7. REFERENCES

[1] Allocation Debugging - The GNU C Library, October 2008. http://www.gnu.org/software/libc/manual/html_node/Allocation-Debugging.html.

[2] M. Bond and K. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2006.

[3] M. Chabbi. Efficient taint analysis using multicore machines. Master's thesis, University of Arizona, 2007.

[4] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

[5] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[6] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[7] W. DePauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 116–134, 1999.

[8] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 USENIX Winter Technical Conference*, pages 125–136, 1992.

[9] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[10] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, 2003.

[11] D. Heine and M. Lam. Static detection of leaks in polymorphic containers. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 252–261, 2006.

[12] M. Jump and K. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–38, 2007.

[13] Mac OS X Developer Tools Manual Page For Leaks, August 2008. http://developer.apple.com/documentation/Darwin/Reference/Manpages/man1/leaks.1.html.

[14] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *WODA '04: Proceedings of the Second International Workshop on Dynamic Analysis*, pages 25–31, 2004.

[15] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 198–209, 2004.

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

[17] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 351–377, 2003.

[18] Mozilla Bugzilla, October 2008. https://bugzilla.mozilla.org/.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[20] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 397–407, 2009.

[21] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *SAS '06: Proceedings of The 13th International Static Analysis Symposium*, 2006.

[22] Parasoft Insure++, August 2008. http://www.parasoft.com/jsp/products/home.jsp?product=Insure.

[23] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, 2005.

[24] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO '08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 74–83, 2008.

[25] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[26] Standard Performance Evaluation Corporation. SPEC CINT2000, October 2008. http://www.spec.org/cpu/CINT2000/.

[27] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. *SIGSOFT Software Engineering Notes*, 30(5):115–125, 2005.

[28] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 151–160, 2008.