AutoFix: An Automated Approach to Memory Leak Fixing on Value-Flow Slices for C Programs

Hua Yan
University of New South Wales, Australia
huayan@cse.unsw.edu.au

Yulei Sui
University of New South Wales, Australia
y.sui@unsw.edu.au

Shiping Chen
Data61
CSIRO, Australia
shiping.chen@csiro.au

Jingling Xue
University of New South Wales, Australia
j.xue@unsw.edu.au

ABSTRACT

C is the most widely used programming language for developing embedded software, operating systems, and device drivers. Unlike programs written in managed languages like Java, C programs rely on explicit memory management, and are therefore prone to memory leaks. Existing (static or dynamic) debugging tools only report leaks, but fixing them often requires considerable manual effort by inspecting a list of reported true and false alarms. How to develop on-demand lightweight techniques for automated leak fixing without introducing new memory errors remains challenging.

In this paper, we introduce AutoFix, a fully automated leak-fixing approach for C programs by combining static and dynamic program analyses. Given a leaky allocation site reported by a static memory leak detector, AutoFix performs a graph reachability analysis to identify leaky paths on the value-flow slices of the program, and then conducts a liveness analysis to locate the program points for inserting fixes (i.e., the missing `free` calls) on the identified leaky paths. We have implemented AutoFix in LLVM-3.5.0 and evaluated it using five SPEC2000 benchmarks and three open-source applications. Experimental results show that AutoFix can safely fix all the memory leaks reported by a state-of-the-art static memory leak detector with small instrumentation overhead.

CCS Concepts

•Software and its engineering → Memory management; Software performance; Software reliability;

Keywords

Memory Leaks; Bug Fixing; Value-Flow Analysis

1. INTRODUCTION

In software testing, the two central tasks facing software engineers are finding bugs and fixing them. Both tasks are expensive in dollar terms and time-consuming due to the ever-increasing scale and complexity of modern software systems. A large number of existing program analyses and testing techniques focus on automatic bug detection. However, finding bugs is only the first step. Once reported, bugs must still be repaired. Indeed, manually fixing bugs can be a non-trivial and error-prone process, especially for large-scale software.


```
1  void Foo ( ) {
2      char* p = malloc (...); //o
3      char* q = "on stack";
4      fgets(p , ... );
5      free (p);                    ✗
6      if (C1) {
7          char * t = p;
8          p = q;
9          q = t;
10     }
11     printf ("%s", q);
12     free (p);                    ✗
13     if (C1)
14         free (q);                ✓
15     else
16         free (p);
17 }
```

(a) Fixing an intraprocedural leak

```
1  void Bar ( ) {
2      char* p = malloc (...); //o
3      fgets(p , ... );
4      if (C2) {
5          Use (p);
6          free (p);
7      }
8      else {
9          Use (p);
10         free (p);                ✓
11     }
12 }
13
14 void Use (char* x) {
15     printf ("%s", x);
16     free (x);                    ✗
17 }
```

(b) Fixing an interprocedural leak

Figure 1: Incorrect (×) and correct (✓) fixes.

Recently, a few approaches to automatic bug fixing have been proposed to reduce maintenance costs by producing candidate patches for program validation and repairing [1, 11, 15, 22, 24, 37]. For example, ClearView [24] enforces violated invariants to correct buffer overflow and illegal control flow errors by creating patches for binaries. AutoFix-E [37] relies on user specifications and generates repairs using contracts. Pachika [9] infers object behavior models to propose candidate fixes for bugs like null dereferences. GenProg [11] uses genetic programming to repair bugs in legacy code.

Most of the existing automatic approaches for fixing bugs in C programs are related to spatial memory errors such as buffer overflows and null pointer dereferences. Such types of bug can be validated by inserting assertions and repaired

by adding conditional checks to avoid executing the code segment that leads to undesired behaviors (e.g., program crashes) [17, 31, 41].

Memory leaks represent another major category of memory errors, i.e. temporal memory errors, which are more complicated to fix automatically. Unlike a spatial error that can be fixed by adding a conditional check to bypass the point where the spatial error occurs, every leaky path from a leaky allocation site needs to be tracked by inserting an appropriate fix (i.e., a `free` call) without introducing new memory errors.

Figure 1 illustrates how intra- and inter-procedural memory leaks are fixed correctly and incorrectly. Suppose the memory allocated in line 2 in Figure 1(a) is never freed. Adding a fix, `free(p)`, too early in line 5 can cause a use-after-free error in line 11, whereas adding `free(p)` in line 12 at the end of program without considering path correlation may introduce an invalid `free` site for a stack object (when the `if` branch is executed). A correct fix is provided in lines 13 – 16, with the underlying path correlation accounted for correctly. Let us consider now an inter-procedural leak shown in Figure 1(b), where the memory allocated in line 2 is leaked partially along the `else` branch (lines 8 – 11). A simple-minded fix, `free(x)`, which is inserted in line 16 in function `Use`, is incorrect. Without considering correlated calling contexts, this fix may introduce a double-free in line 6 when the `if` branch is executed. A correct fix is given in line 10, ensuring that only the leak along the `else` branch is fixed.

Existing (static and dynamic) memory leak detectors for C programs only report leaks, but fixing them along every leaky path remains to be done manually by programmers. Dynamic detectors [4, 21] find memory leaks by instrumenting and tracking memory accesses at runtime, incurring high overhead. By testing a program under some inputs, dynamic detectors typically compute an under-approximation which produces no false positives but potentially misses many bugs. In contrast, static detectors [6, 16, 32, 33, 38], which over-approximate runtime behaviors of the program without executing it, can soundly pinpoint all the leaks in the program, but at the expense of some false positives.

This paper presents AutoFix, a fully automated approach to fixing memory leaks in C programs by combining static and dynamic analyses. Given a list of leaky allocation sites reported by a static detector, AutoFix automatically fixes all the reported leaks by inserting appropriate fixes (i.e., `free` calls) along all the leaky paths. There are two main challenges. First, a detector reports a leaky allocation site as long as it discovers one leaky path from the site without necessarily reporting all the leaky paths. AutoFix is designed to fully repair the memory leak for all its leaky paths. Second, some reported leaks are false positives. AutoFix must guarantee memory safety by ensuring that the fixes are correct regardless of whether the reported leak is a true bug or a false positive. Note that AutoFix certainly cannot fix any leaks that are missed (i.e., not reported) by a static detector.

AutoFix applies to a large class of real-world C programs where memory management is explicitly orchestrated by

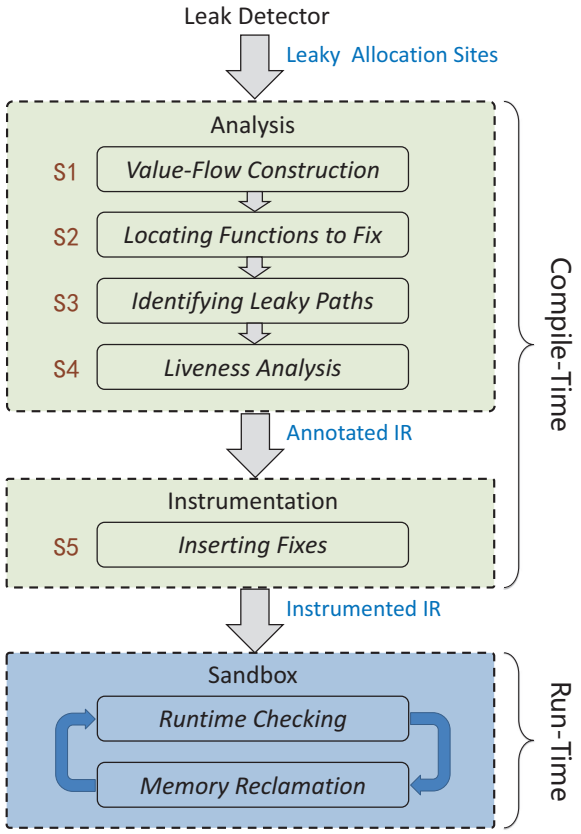


Figure 2: The AutoFix framework.

programmers without resorting to garbage collection (GC) and/or reference counting (RC). Compared to the GC and RC approaches, our approach is lightweight as only small instrumentation overhead is incurred. To safely reclaim a leaked memory object o from an allocation site without any programmer intervention, all the memory allocation and deallocation sites reachable from o on the value-flow slices of the program are instrumented to keep track of the liveness of o in shadow memory, thereby enforcing correct leak fixing inside a memory-safe execution sandbox at runtime.

Figure 2 highlights the basic idea behind AutoFix. Given a leaked object o from an allocation site (reported by any leak detector), AutoFix builds from the program a sparse value-flow graph (S1), on which a graph reachability analysis is first performed to locate the candidate functions for inserting appropriate fixes, i.e., `free` calls (S2). For each candidate function f, AutoFix then identifies the leaky paths in f for o by computing the value-flow guards with respect to its existing deallocation sites found in the program (S3). Next, a liveness analysis is applied inside f on the value-flow slice of the identified leaky paths for o to determine every program point P where a fix is needed with path correlation considered (S4). Finally, the fixes are inserted immediately after the last use sites of o on all its leaky paths (S5). At runtime, the instrumented fixes performs runtime checking to verify statically identified leaks, and reclaims only true leaked memory objects.

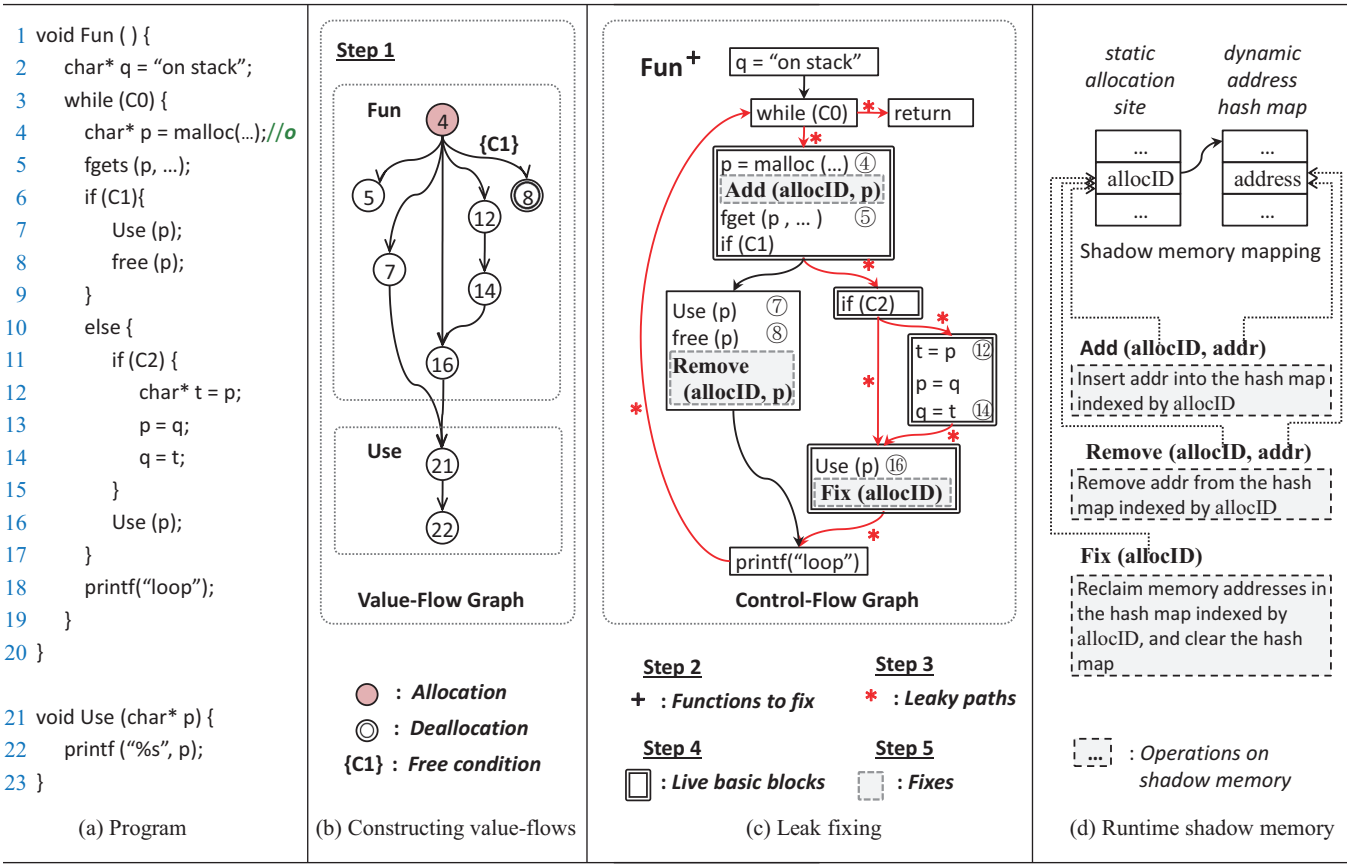


Figure 3: A motivating example.

This paper makes the following contributions:

- We present AutoFix, a fully automated approach to memory leak fixing for C programs that can reclaim **all** the leaked memory objects reported by any leak detector.
- We propose an approach to **safe** memory leak fixing by combining static and dynamic analyses: an inter-procedural static analysis is first performed to identify the earliest program points that the missing `free` calls can be inserted without introducing any use-after-free error; and then, a dynamic analysis, which operates on efficient shadow memory, tracks the potentially leaked memory objects and performs runtime checking to fix true leaks without introducing any double-free error.
- We have implemented AutoFix in LLVM-3.5.0 and evaluated it using five SPEC2000 benchmarks and three open-source applications. Our experimental results show that AutoFix can safely fix all the leaks reported by the state-of-the-art static leak detector, Saber [32, 33], with runtime overhead averaged at under 2%. For the long-running server application `redis` evaluated, AutoFix has successfully reduced its memory usage by more than 300MB in a three-hour continuous run after having fixed its leaks.

2. A MOTIVATING EXAMPLE

In order to describe the main idea of our approach, we use the example in Figure 3 to go through the five key steps shown in Figure 2. An allocation site in line 4 of `fun` in Figure 3(a) is partially leaky along the `else` branch in line 10 inside a `while` loop. Given a leaked object o detected from this allocation site by a static leak detector, AutoFix first constructs a value-flow graph shown in Figure 3(b) for o. Based on this graph, AutoFix inserts one fix immediately after line 16 on the leaky `else` branch, as shown in Figure 3(c). The leaky allocation site (in line 4) and the deallocation site (in line 8) in the non-leaky `if` branch are instrumented with dynamic checks to ensure safe fixing at runtime, as illustrated in Figure 3(d).

Step 1: Constructing Value-Flows. Following [32, 33], we construct an inter-procedural value-flow graph (VFG) for every leaky allocation, with the one for the example given in Figure 3(b). Note that its nodes are numbered by their corresponding line numbers in Figure 3(a). In a VFG, its nodes represent the definitions of variables and its edges capture their def-use relations.

Step 2: Locating Functions to Fix. Given a leaked object o from an allocation site, AUTOFIX first determines the functions where the leaks of o should be fixed. A candidate function f is chosen if f allocates o (directly in itself or indirectly in its callee functions) such that o is never returned to any caller of f. Note that the existence of a candidate function for o is guaranteed since the `main` function will be the last resort. In our example, `Fun` is selected as a candidate function since it contains an allocation site of o (line 4) and o is never returned to any callers of `Fun` based on its value-flows.

Step 3: Identifying Leaky Paths. AUTOFIX identifies the leaky paths for o in `Fun` by reasoning about value-flow guards, which are Boolean formulas capturing branch conditions between defs and uses in the control flow graph (CFG). The *free* condition $C1$ under which the `free` site in line 8 is reached is computed by performing a guarded reachability analysis from the `malloc` source ④ to its `free` sink ⑧. Thus, the *leak* condition $\neg C1$ encodes the leaky paths in the `else` branch as highlighted in red in Figure 3(c).

Step 4: Liveness Analysis. An intra-procedural liveness analysis is performed for `Fun` to mark the live basic blocks for o (shown as double-framed boxes in Figure 3(c)) that are reachable from the allocation site of o on its leaky paths. As a result, node ⑯ is identified as the last-use site of o.

Step 5: Inserting Fixes. As shown in Figure 3(c), a deallocation Fix() is inserted immediately after ⑯ (i.e., line 16), where the last use of o is found. In addition, the instrumentation code (in dotted-line boxes) also includes the metadata-manipulating functions inserted (after the `malloc` source ④ and the `free` sink ⑧) to maintain runtime shadows for o to ensure safe fixing for both leaky and non-leaky paths.

Figure 3(d) shows that the shadow memory simply maps an allocation site with its unique ID, `allocID`, to a hash map that records (start) addresses of the dynamically allocated objects that are not yet freed. Consider Figure 3(c) again. Every address p that points to an object allocated at ④ is recorded in the shadow memory by calling function Add(`allocID`, p) instrumented immediately after ④. The deallocation site ⑧, which is reachable from ④ via value-flows, is instrumented by calling function Remove(`allocID`, p) to delete the address p from the shadow hash map since its pointed-to object has been released along the non-leaky `if` branch. On reaching Fix(`allocID`) during program execution, all the objects identified by the addresses corresponding to the allocation site `allocID` in the shadow memory are freed (as they would be leaked otherwise).

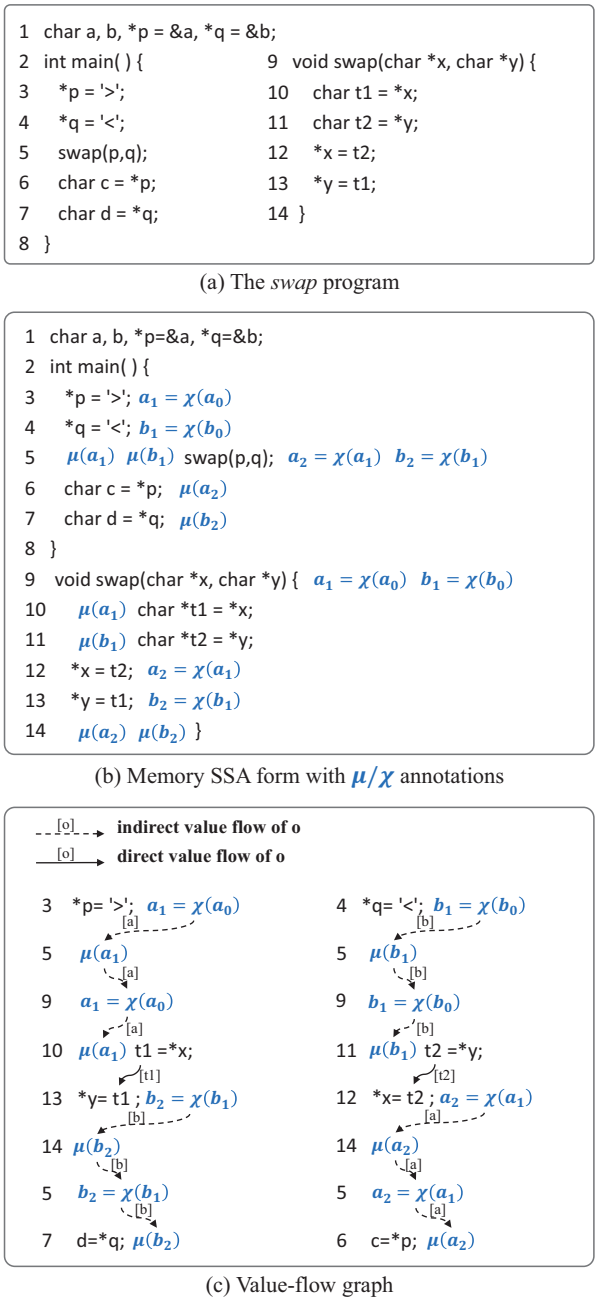


Figure 4: Value-flow example.

3. AUTOMATED LEAK FIXING

AUTOFIX is a compile-time transformation for inserting runtime checks to reclaim leaked memory for C programs, which keeps track of potential leaked memory addresses via an efficient shadow metadata structure. In this section, we first present the five steps of AUTOFIX's compile-time transformation (§3.1 – §3.5), and then describe the design of AUTOFIX's metadata structure (§3.6).

3.1 Step 1: Constructing a Value-Flow Graph

An inter-procedural sparse value-flow graph (VFG) [12, 29, 32, 33, 34, 35, 42, 43] for a program is a multi-edged directed graph that captures the def-use chains of both top-level and address-taken variables. Top-level variables are the variables whose addresses are not taken. The def-use chains for top-level variables are readily available once they have been put in SSA form as is standard. Address-taken variables are accessed indirectly and inexplicitly at loads and stores. Their def-use chains are built in several steps following [7, 32, 33].

Algorithm 1: Liveness Analysis (for leaked object o)

```
1  Let F be the set of candidate functions for fixing o
2  foreach f ∈ F do
3      Let subCFG be the subgraph of f's CFG that
       contains only the leaky paths for o in f
4      foreach basic block b in subCFG do
5          isLive(b) ← false
6          if b contains an o-reachable variable then
7              isLive(b) ← true
8      while isLive has changed do
9          foreach basic block b in subCFG do
10             isLive(b) ← ⋁_{s∈succ(b)} isLive(s)
11             // succ(b) is the set of successors of b
```

First, the points-to information for the program is computed by using, e.g., Andersen's analysis. Second, the accesses of address-taken variables are made explicit by adding annotations. A load $p = *q$ is annotated with a function $\mu(x)$ for each variable x that may be pointed to by q to represent a potential use of x at the load; A store $*p = q$ is annotated with $x = \chi(x)$ for each variable x that may be pointed to by p to represent a potential def and use of x at the store; A call site cs is also annotated with $\mu(x)/x = \chi(x)$ for each variable x that is inter-procedurally referred/modified inside the callee of cs. A function definition def `f`(...) is annotated at the entry/exit with $\mu(x)/x = \chi(x)$ for each variable x that is inter-procedurally referred/modified inside f. Third, all the address-taken variables are converted to SSA form, with each $\mu(x)$ being treated as a use of x and each $x = \chi(x)$ as both a def and use of x. Finally, the value-flows are constructed by connecting the def-uses for each converted SSA variable.

Figure 4 gives an example. The program in Figure 4(a) defines two variables a and b in line 1, whose addresses are taken by pointers p and q, respectively. By indirectly memory accesses via pointers p and q, a and b are first initialized in line 3 and 4 respectively, and are then passed to subroutine `swap` in line 5. Finally, c gets a's value ('<' after the swap) by dereferencing p in line 6, while d gets b's value ('>' after the swap) by dereferencing q in line 7. To track indirect value-flows, annotations are added to make indirect memory accesses explicit, as shown in Figure 4(b), where the subscripts of a and b are as in standard SSA form. Note that the subscripts start from zero at each function entry (a_0 in line 3 for `main` and in line 9 for `swap`, b_0 in line 4 for `main` and in line 9 for `swap`). By connecting def-uses based on the annotations, the value-flow graph in Figure 4(c) is constructed, which shows how a and b's values are exchanged and then flow to c and d respectively.

3.2 Step 2: Locating Functions to Fix

Definition 1 (Value-flow Reachability). *A variable v is o-*reachable *if there exists a value-flow path from the allocation site of o to the definition site of v on the VFG of the program. A callsite $p = call(...)$ is o-*reachable *if either variable p or x in any $x = \chi(x)$ function annotated at the callsite is o-reachable.*

Given a leaked object o from an allocation site reported by a static detector, AutoFix first determines the candidate functions where the leaks of o will be fixed. A function f is a candidate function to insert fixes for o if (1) f contains at least one o-reachable callsite and (2) there is no o-reachable variable in any caller of f.

In the case of recursion, if there is no data dependence on the leaked object o between any two function calls in the recursive cycle, o can be fixed in the recursive functions; otherwise fixes for o must be put outside the recursive functions to ensure safe fixing. In the case of global variables, since they are reachable for every function, leaked objects pointed to by global pointers can only be fixed in `main`. However, global variables are generally not considered as leaks in existing leak detectors [6, 16, 32, 33].

3.3 Step 3: Identifying Leaky Paths

To identify the leaky paths in a candidate function f, AutoFix performs a forward analysis on the VFG from an o-reachable callsite src to construct a value-flow slice S_{src} that includes all the nodes reachable from src but confined in f.

If no `free` sites are reachable from src, then all paths in function f are leaky paths. Otherwise, for a `free` site snk corresponding to S_{src}, let *vfp(src, snk)* be the set of all value-flow paths from src to snk on the VFG, and *vfe(P)* be the set of all value-flow edges in a single value-flow path $P \in vfp(src, snk)$. Thus, we can obtain the value-flow guards from src to snk:

$$VFGuard(src, snk) = \bigvee_{P \in vfp(src,snk)} \bigwedge_{(\hat{s},\hat{d}) \in vfe(P)} CFGuard(\hat{s}, \hat{d})$$

$$CFGuard(\hat{s}, \hat{d}) = \bigvee_{Q \in cfp(\hat{s},\hat{d})} \bigwedge_{e \in Q} boolCond(e)$$

where $CFGuard(\hat{s}, \hat{d})$ is a boolean formula that encodes the set of control-flow paths, denoted as $cfp(\hat{s}, \hat{d})$, from program point $\hat{s}$ to $\hat{d}$ on f's CFG. Each branch edge e on a control flow path Q is uniquely assigned a boolean variable $boolCond(e)$. In the presence of loops, guards can grow unboundedly. To avoid unbounded conjunctions that describe all loop iterations, we follow [6, 32, 33] and bound loops to one iteration. Finally, the leak condition for src is obtained by computing guards from src to all its reachable `free` sites in f:

$$LeakCond = \overline{\bigvee_{snk \in S_{src}} VFGuard(src, snk)}$$

Any path from src to the end of function f that satisfies $LeakCond$ is a leaky path for the leaked object o.

3.4 Step 4: Liveness Analysis

For a candidate function f, a subgraph is extracted from f's CFG by excluding the control flow edges that are not on any leaky path. Then, a liveness analysis is applied to this

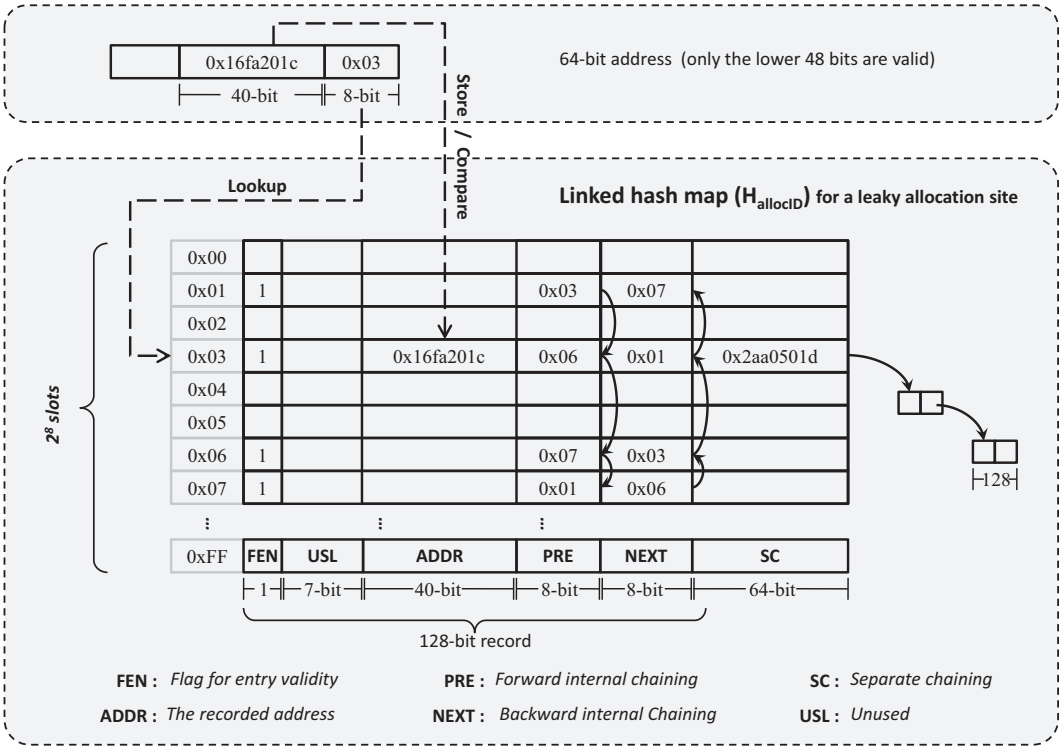


Figure 5: The metadata structure design.

subgraph to determine the basic blocks in f where o may be live.

As shown in Algorithm 1, a backward data-flow analysis is performed in f, starting from blocks containing o-reachable variables (lines 6 – 7) and iteratively marking the liveness of each block until a fix-point is reached (lines 8 – 11). For a leaked object o allocated inside a loop, if there is no data dependence on o between different loop iterations, then fixes for o can be inserted inside the loop; otherwise fixes for o must be put outside the loop to ensure safe fixing.

3.5 Step 5: Instrumentation

Based on the liveness information, Algorithm 2 performs instrumentation to insert fixes for every leaked object o in each of its candidate functions f. A call to Fix() is inserted either at the end of b where b is the last live basic block for o (lines 6 – 7), or at the beginning of n where n is a newly created basic block between a live block and a non-live block of o (lines 9 – 11). As shown earlier in Figure 3(d), these fixes serve to reclaim the dynamically allocated memory at the allocation site of o that would otherwise be leaked. In addition, calls to Add() and Remove() are instrumented to maintain runtime shadow memory, thereby enforcing safe leak-fixing. A call to Add() is inserted after the allocation site of o (line 12) to track all its allocated objects in the shadow memory. A call to Remove() is inserted after each `free` site reachable from its corresponding allocation site (line 13), so that the freed objects are removed from the shadow memory.

Our instrumentation is safe even if the leaked object o is a false positive, for two reasons. First, the VFG of a program over-approximates its def-use chains. Thus, the last-use sites of o in its candidate functions for fixing o are conservatively found, ensuring safety by avoiding any use-after-free. Second, Add() and Remove() maintain valid memory addresses in the shadow memory, ensuring safety by avoiding any potential double free along any program path.

Algorithm 2: Instrumentation (for leaked object o)

```
1  Let F be the set of candidate functions for fixing o
2  Let allocID be the unique ID of o's allocation site
3  foreach f ∈ F do
4      Let liveBBs be the set of basic blocks that are
       marked live for o in f's CFG
5      foreach b ∈ liveBBs do
6          if ∄s ∈ succ(b), isLive(s) = true then
7              Insert Fix(allocID) at the end of b
8          else
9              foreach s ∈ succ(b) s.t. isLive(s) = false
               do
10                 Insert a new block n between b and s
11                 Insert Fix(allocID) at the beginning of n
12 Insert Add(allocID, p) immediately after the
   allocation site p = malloc(...) for o
13 Insert Remove(allocID, q) immediately after each free
   site free(q) where q is o-reachable
```

3.6 The Metadata Structure

The design philosophy behind our metadata structure is to enable a judicious tradeoff between time and space, which aims to support fast lookup, insertion and removal operations with reasonable space overhead. As shown in Figure 5, AutoFix maintains a closed hash map $H_{allocID}$ for every leaky allocation site (with its unique ID, `allocID`) to keep track of all the dynamic allocated memory addresses. The size of $H_{allocID}$ can be user-defined, with larger hash maps consuming more space and smaller ones potentially imposing higher slowdown due to hash collisions. To achieve a reasonable tradeoff (as evaluated in §4.3), the default hash map size is set to 2^8, with 128 bits for each slot, resulting in a total of 4 KB consumed for the hash map.

Without loss of generality, our shadow mechanism is supported on a 64-bit x86_64 architectures with 48-bit virtual address space and word-aligned pointers. For a 64-bit memory address allocated at a leaky site, AutoFix uses its lower 8 bits as the index to the corresponding entry in the hash map, and maps its middle 40 bits to the field ADDR of the entry. A linked list is implemented to handle hash collisions in each hash slot, with the field SC recording the head of the list. Due to the sparsity of the hash map, it is expensive to retrieve all the valid entries to reclaim leaked memory by performing a full scan for the map. To speed up the search, we have used a doubly linked list (similar to Java's `LinkedHashmap`) with the two 8-bit fields, PRE and NEXT, to record the previous and the next valid entry indexes. The one-bit field FEN indicates whether an entry is valid or not. It is set to 0 when the entry is removed from the hash map. The 7-bit field USL is preserved for future use.

Figure 6 gives the implementation of our shadow metadata functions i.e. Add(), Remove(), Fix(). The *lookup*, *insert*, *remove* and *clear* are standard operations that are similar to those in Java's `LinkedHashmap` and are thus omitted. Function Add(), which is instrumented immediately after the leaky allocation site `allocID`, first finds a slot in $H_{allocID}$ to create an entry for the allocated memory object o (line 1), then maps bits 8 through 47 of o's address $addr_o$ to the 40-bit field ADDR using a simple shift operation (line 2), and finally inserts the entry into $H_{allocID}$ (line 3). Function Remove(), which is instrumented after a deallocation site of o, checks whether the deallocated address is recorded in $H_{allocID}$ (lines 4 and 5). If so, the corresponding entry is removed (line 6). Function Fix(), which is instrumented after the last-use site of o, first traverses $H_{allocID}$ using its internal linked list via *getNext* (lines 7), then frees all the recorded addresses (line 8) and clears the hash map (line 9).

```
// o's allocation site: allocID
addr_o = malloc(...);
1 entry = H_allocID -> lookup(addr_o);
2 entry -> ADDR = (uint64_t)addr_o >> 8;
3 H_allocID -> insert(entry);
```

(a) Add(allocID, $addr_o$)

```
// o's deallocation site
free(addr_o);
4 entry = H_allocID -> lookup(addr_o);
5 if (entry)
6       H_allocID -> remove(entry);
```

(b) Remove(allocID, $addr_o$)

```
// o's last-use site
lastUse(addr_o);
7 for(entry = H_allocID -> getHead(); entry != 0;
  entry = H_allocID -> getNext(entry))
8       free(entry -> ADDR << 8| entry - H_allocID);
9 H_allocID -> clear();
```

(c) Fix(allocID)

Figure 6: Implementation of shadow metadata operations.

4. EVALUATION

We have implemented AutoFix on top of LLVM (version 3.5.0). Eight C programs are used for evaluation as shown in Table 1, including five SPEC2000 benchmarks and three popular open-source applications. The SPEC2000 suite is widely used for evaluating static leak detectors [6, 16, 32, 33]. However, the SPEC2000 benchmarks that have less than two reported leaks (e.g. `parser` and `gap`) are excluded from our evaluation. The five selected SPEC2000 benchmarks are `ammp` (contains many leaks), `gcc` (large-sized and contains many leaks), `perlbmkm` (allocation-intensive), `twolf` (allocation-intensive) and `mesa` (deallocation-intensive). We also include three open-source applications: `a2ps-4.14` (a postscript filter) containing a relative large number of leaks, and two long-running server programs: `h2o-1.2` (an http server) and `redis-2.8` (a NoSQL database).

All our experiments are conducted on a platform consisting of a 3.0 GHZ Intel Core2 Duo processor with 16 GB memory, running RedHat Enterprise Linux 5 (kernel version 2.6.18). The source code of each program is compiled into bit-code files using clang and then merged together using LLVM Gold Plugin at link-time (LTO) to produce a whole-program bit-code file. The compiler option mem2reg is turned on to promote memory into registers. Andersen's pointer analysis is used to build the VFG for the program [12]. We use the leak warnings (leaky allocation sites) reported by the state-of-the-art leak detector, Saber [32, 33], as input to AutoFix.

We evaluate AutoFix based on three criteria: (1) efficiency (number of fixes generated and the analysis time taken to do so), (2) effectiveness (ability to fix memory leaks and reduce memory usage at runtime), and (3) performance degradation (instrumentation overhead at runtime).

4.1 Efficiency of AutoFix

The compile-time results of AutoFix are summarized in

Table 1: Benchmark characteristics

Program	Size (KLOC)	#Alloc Sites	#Free Sites	#Leaky Alloc Sites Reported	#True Leaks
ammp	13.4	37	30	20	20
gcc	230.4	161	19	45	40
perlbmk	87.1	148	2	12	8
mesa	61.3	82	76	7	3
twolf	20.5	185	1	5	5
a2ps	41.8	295	161	39	28
h2o	18.2	95	123	27	26
redis	61.8	47	62	24	20
Total	534.5	1050	474	179	150

```
1   char* p = "..."; // stack obj
2   if (...) {
3       p = malloc(...); // heap obj
4       ...
5   }
6   use(p); // conservative last use site of heap obj
```

(a) Code with a memory leak before applying AutoFix

```
1   char* p = "..."; // stack obj
2   if (...) {
3       p = malloc(...); // heap obj
++      Add(...); // instrumented by AutoFix
4       ...
5   }
6   use(p); // conservative last use site of heap obj
++  Fix(...); // instrumented by AutoFix
```

(b) Code without memory leaks after applying AutoFix

Figure 7: Fixing path correlated leaks in AutoFix.

Table 2. Given a total of 179 leaky allocations reported in the eight programs, AutoFix fixes them all by inserting 393 calls to Fix() (Column 2), 179 calls to Add() (Column 3) and 107 calls to Remove() (Column 4). On average, a leaky allocation results in only 2.2 fixes. This shows that AutoFix is able to precisely place fixes along the identified leaky paths with lightweight instrumentation. As shown in Table 2 (Column 5), it takes 216.1 seconds to analyze the 534.5 KLOC for the eight C programs altogether. In particular, AutoFix spends 81.7 seconds on `gcc`, the largest program (230.4 KLOC) studied. The analysis times for the other seven programs are all within one minute.

4.2 Effectiveness of AutoFix

To evaluate the effectiveness of AutoFix in fixing leaks at runtime, we compare the memory usage of each program before and after automated fixing using Valgrind [21]. For the five `SPEC2000` benchmarks, their *reference* inputs are

```
256  static int _redisContextConnectTcp(...) {
         ...
281      for (p = servinfo; p != NULL; p = p->ai_next) {
           ...
291        rv = getaddrinfo(..., &bservinfo); // o's allocation site
++         Add (bservinfo, allocID); // instrumented by AutoFix
297        for (b = bservinfo; b != NULL; b = b->ai_next) {
               ...     // o is used in the loop
302        }
++         Fix (allocID);              // instrumented by AutoFix
329    }
342  }
```

Figure 8: The leaky code (in net.c of redis-2.8) fixed by AutoFix.

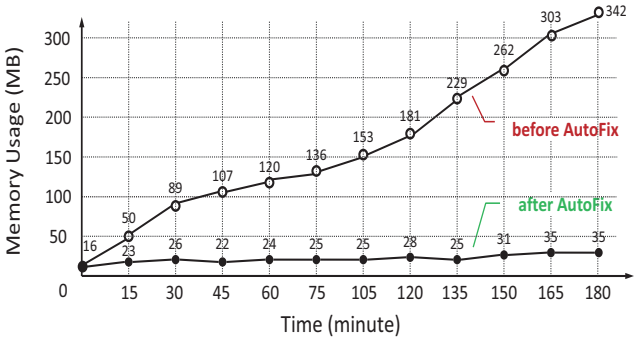


Figure 9: Memory footprint of redis-2.8 before and after fixing the leak.

used. For the three open-source applications, their own regression test suites are used. For a total of 67 real leaks triggered by the inputs (Column 2 in Table 3), AutoFix is able to reclaim all the leaked memory at runtime, which is verified by Valgrind [21].

In our experiments, we observed that a substantial number of leaks are inter-procedural, involving path correlation. These leaks are ignored and cannot be fixed by the pure static approach LeakFix [10] due to the over-approximative nature of static analysis. In contrast, AutoFix combines static analysis with runtime checking to enable precise fixing for all leaks including those involving path correlations.

Figure 7 shows a memory leak pattern in `gcc` and the instrumented code with the leak fixed. The pointer p at the callsite `use` (in line 6) may point to either (1) a heap object (allocated in line 3) when the `if` branch is taken, or (2) a stack object (allocated in line 1) otherwise. A leak happens in the former case, while the code is leak-free in the latter case. AutoFix tracks the memory allocation by instrumenting an Add after `malloc` (in line 3) and reclaims only truly leaked memory by performing runtime checks in Fix immediately after the last use site (in line 6) conservatively computed by value-flow analysis.

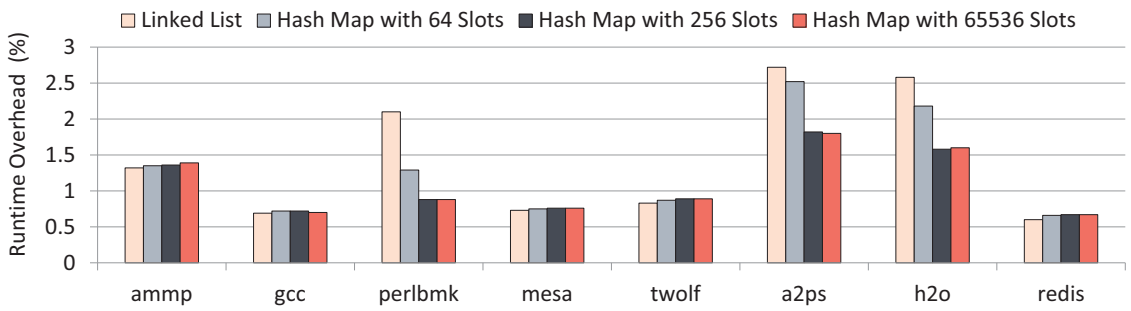


Figure 10: Comparing runtime overheads for different metadata structures used in AutoFix.

Table 2: Compile-time statistics of AutoFix

Program	#Fix()	#Add()	#Remove()	Analysis Time(s)
ammp	20	20	0	0.9
gcc	74	45	13	81.7
perlbmk	131	12	28	32.0
mesa	19	7	9	15.1
twolf	7	5	0	3.9
a2ps	51	39	48	17.0
h2o	61	27	2	9.5
redis	30	24	7	56.0
Total	393	179	107	216.1

Table 3: Run-time statistics of AutoFix

Program	#Triggered Leaks	Overhead (%)
ammp	1	1.36
gcc	13	0.75
perlbmk	12	0.88
mesa	3	0.76
twolf	2	0.89
a2ps	12	1.82
h2o	15	1.58
redis	9	0.66

To further evaluate the effectiveness of AutoFix in fixing leaks for long-running programs, we reproduce a real leak which causes memory exhaustion in `redis` with its corresponding regression tests [20]. As shown in Figure 8, a loop (line 281) is used to query the IPs of slave servers. If a slave sever is dead, reconnection attempts are repeatedly made by calling `getaddrinfo` (line 291), which allocates memory chunks that are never freed, resulting in leaks inside the `for` loop. As shown in Figure 9, the memory consumption of the leaked version of `redis` increases around 31.7 KB per second, and over 300 MB are leaked after three hours. AutoFix can fix this leak effectively, enabling `redis`'s memory consumption to remain below 35 MB in the fixed version.

4.3 Runtime Overhead

To measure runtime overhead, each program is executed five times before and after automated fixing respectively, and the average overhead is reported in Table 3. The experimental results show that AutoFix only introduces negligible overhead for all the eight programs, 1.06% on average, with the maximum 1.82% observed in `a2ps`. This confirms that our instrumentation is lightweight, achieved by identifying the required deallocation fixes on the value-flow slices of leaky allocations and tracking leaked objects with simple shadow operations at runtime.

To evaluate the impact of the metadata structure (as described in §3.6) on runtime instrumentation overhead, we choose four different sizes for the hash map used in order to demonstrate the time and space tradeoffs made: 1 (with the hash map degenerating into one linked list), 2^6, 2^8 and 2^{16}. The results are shown in Figure 10.

For the five benchmarks, `ammp`, `gcc`, `mesa`, `twolf` and `redis`, the four configurations yield similar overheads. However, for the other three benchmarks, `perlbmk`, `a2ps` and `h2o`, much higher overheads are incurred when their underlying hash maps have degenerated into a single linked list. In this degenerate case, the lookup operations become too expensive, especially when a large number of memory objects are present. When the other three hash map sizes are used, lookup operations can be performed more efficiently. The hash map with 2^6 slots is not very space-consuming, costing 1 KB for each leaky allocation. However, due to its high collision rates, this hash map still results in high overheads for `perlbmk`, `a2ps` and `h2o`. As shown in Figure 10, the hash maps with 2^8 slots (4 KB per leaky allocation) and 2^{16} slots (1 MB per leaky allocation) suffer from similar overheads. This indicates that 2^8 slots are already sufficient to guarantee low collision rates, and more slots cannot provide any noticeable performance benefit. For more complicated applications beyond our evaluation, it is still possible that 2^8 slots are not enough to ensure low hash collision rates. In this situation, AutoFix allows users to allocate more slots for the shadow hash map to achieve better performance.

To understand how the runtime overhead are caused and distributed, we profile each Add, Remove and Fix operation instrumented. Figure 11 shows the result. For all the programs evaluated, maintaining shadow memory (Add and Remove together) incurs more overhead than reclaiming the leaked memory (Fix) by an average of 31.3%. For the two programs without Remove operation instrumented (i.e. `ammp` and `twolf`), the Add operation alone (63% in `ammp` and 58% in `twolf`) still accounts for more overhead than the

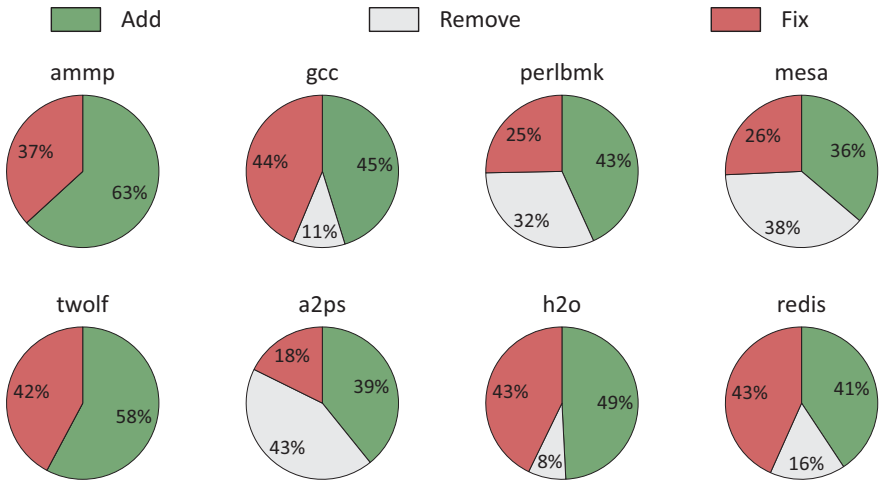


Figure 11: Runtime overhead breakdown.

FIX operation (37% in `ammp` and 42% in `twolf`). The REMOVE operation causes significant proportion of overhead in `perlbmk` (32%), `mesa` (38%), and `a2ps` (43%). These three programs also have relatively high numbers of false memory leaks, as shown in Table 1. AUTOFIX has to identify and tolerate false memory leaks by the REMOVE operation when fixing true ones.

5. RELATED WORK

Leak Detection: Memory leak detection has been extensively studied using static [16, 32, 33, 38] or dynamic [13, 21, 27] analysis. Static detectors examine the source code at compile-time without executing the program. SATURN [38] detects memory leaks by solving a Boolean satisfiability problem. SPARROW [16] is based on abstract interpretation, and uses function summaries. FASTCHECK [6] and SABER [32, 33] find memory leaks on the value-flow graph of the program. Dynamic detectors, which find leaks by executing the program, track the memory allocation and deallocation via either binary instrumentation as in VALGRIND [21] or source code instrumentation as in ADDRESSSANITIZER [27].

Leak Tolerance: Another line of research focuses on tolerating leaks at runtime [3, 23, 36]. The basic idea is to delay out-of-memory crashes at runtime by offloading stale objects (regarded as likely leaked) to disks and reclaiming their virtual memory. Upon accessing a mistakenly swapped-out object, the object will be swapped back into the memory, thereby guaranteeing safety. Apart from the space overhead, dynamically detecting stale objects by tracking accesses of memory objects also results in non-negligible time overhead. For example, LeakSurvivor [36] incurs an average runtime overhead of 23.7% even for applications without memory leaks. AUTOFIX, which aims at fixing leaks, is orthogonal to tolerating leaks. Instead of dynamically tracking every memory access of every object to determine objects' liveness, AUTOFIX conservatively approximates the liveness for only leaky objects at compile-time, therefore avoiding high runtime overhead.

Garbage Collection: Garbage collection (GC) can eliminate most memory leaks. However, in type-unsafe languages such as C and C++, it is theoretically impossible to implement sound GC to automatically manage memory. A few unsound (conservative) solutions for C and C++ [2, 14, 25] have been shown empirically to be effective with low space and time overheads, in which memory allocations (e.g. `malloc` sites) are replaced by special allocators, and memory deallocations (e.g. `free` sites) are removed from the program, at the expense of the prompt low-cost reclamation provided by explicit memory management. Compared to AUTOFIX's static fixing on value-flow slices, GC uses runtime object reachability to over-approximate object liveness. In addition, garbage collectors for C and C++ typically need to monitor all static data areas, stacks, registers and heap. In contrast, AUTOFIX only monitors potential leaky allocations reported by leak detectors, which makes AUTOFIX much more lightweight than GC. AUTOFIX and conservative GC can be applied simultaneously, with the former in charge of the leaked memory objects allocated by `malloc` and the latter in charge of the memory allocated by GC's special allocators.

Leak Fixing: Memory leaks can be fixed manually or automatically. LEAKPOINT [8] is a dynamic taint analysis that identifies last-use sites of leaked objects by tracking pointers and presents programmers the identified sites as candidate locations for leak fixing. LEAKCHASER [39] relies on user annotations to improve the relevance of bug reports, thereby assisting programmers to diagnose and fix memory leaks. Object ownership profiling has also been applied to assisting manual leak detection and fixing [26]. LEAKFIX [10] is a pure static approach to automatically fixing leaks in C programs. Because it cannot handle false positives produced by other state-of-the-art leak detectors, LEAKFIX relies on its own dedicated leak detector and can fix only some but not all reported leaks. In contrast, our approach combines

static and dynamic analyses, and is able to automatically fix all the true leaks reported by a detector with small runtime overhead.

Value-Flow Analysis: Value-flow analysis computes inter-procedural def-use information for both top-level and address-taken variables. A prerequisite for value-flow analysis is the pointer/alias information provided by pointer analysis. Value-flow analysis, in turn, can assist pointer analysis to improve precision and scalability [12, 18, 19, 30]. Recently, value-flow analysis has been applied in memory error detection [6, 32, 33, 42], program slicing [28], and inter-procedural SSA analysis [5].

6. DISCUSSION

Like many program analysis and software testing problems, static memory leak detection and fixing are undecidable in general. At its core, it is undecidable whether each leaky path is feasible. Moreover, even if we ignore runtime evaluation of branch conditions and assume all static control-flow paths are feasible, it is still impossible to develop an approach that can statically fix all memory leaks by only inserting missing `free` calls. This is because it requires the following strict condition to be satisfied for each leaky path ρ of each leaked object o: there must exist a program point l on ρ and l must not be on any non-leaky path ρ'. If the condition is not satisfied, the `free` call instrumented may mistakenly free non-leaked memory objects and is therefore unsafe. However, the condition has been proven to be not satisfiable for many memory leaks [40]. As a result, pure static approach (e.g. LeakFix [10]) can fix only some memory leaks.

The current AutoFix implementation reclaims leaked memory as early as possible, which minimizes performance degradation caused by memory leaks. However, this is not necessarily the solution with the smallest instrumentation overhead, because postponing a fix may allow multiple instrumentations to be merged, there by reducing runtime overhead. We leave this as our future work.

7. CONCLUSION

This paper presents AutoFix, a fully automated approach to memory leak fixing for C programs by combining static and dynamic analysis. Given a leaky allocation reported by a leak detector, AutoFix performs a graph reachability analysis to identify the leaky paths on the value-flow graph of the program, and then performs a liveness analysis to locate the program points for instrumenting the required fixes on the identified leaky paths at compile-time. To guarantee safe fixing, shadow memory is maintained for the potential leaked memory objects at runtime. Our evaluation shows that AutoFix is capable of fixing all reported memory leaks with small instrumentation overhead.

8. ACKNOWLEDGEMENTS

This work is supported by ARC grants, DP130101970 and DP150102109.

9. REFERENCES

[1] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE World Congress on Computational Intelligence*, pages 162–168, 2008.

[2] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *POPL '02*, pages 93–100, 2002.

[3] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *OOPSLA'08*, pages 109–126, 2008.

[4] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *CGO '11*, pages 213–223, 2011.

[5] S. Calman and J. Zhu. Increasing the scope and resolution of interprocedural static single assignment. In *SAS'09*, pages 154–170, 2009.

[6] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*, pages 480–491, 2007.

[7] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267, 1996.

[8] J. Clause and A. Orso. Leakpoint: pinpointing the causes of memory leaks. In *ICSE '10*, pages 515–524, 2010.

[9] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE '09*, pages 550–554, 2009.

[10] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In *ICSE'15*, pages 459–470, 2015.

[11] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. on Software Engineering*, 38(1):54–72, 2012.

[12] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*, pages 289–298, 2011.

[13] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS '04*, pages 156–164, 2004.

[14] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 24(6):593–624, 2002.

[15] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI '12*, pages 221–236, 2012.

[16] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *ISMM'08*, pages 131–140, 2008.

[17] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[18] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*, pages 343–353.

[19] L. Li, C. Cifuentes, and N. Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM'13*, pages 85–96, 2013.

[20] D. Mezzatto. Sentinel 2.8 branch memory leak in redis, https://github.com/antirez/redis/issues/2012, 2014.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100, 2007.

[22] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *ICSE '13*, pages 772–781, 2013.

[23] G. Novark, E. D. Berger, and B. G. Zorn. Plug: automatically tolerating memory leaks in C and C++ applications. *Technical Report UM-CS-2008-009, University of Massachusetts*, 2008.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *SOSP '09*, pages 87–102, 2009.

[25] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for c. In *ISMM '09*, pages 39–48, 2009.

[26] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *ASE'07*, pages 194–203, 2007.

[27] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[28] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI '07*, pages 112–122, 2007.

[29] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multithreaded C programs. In *CGO'16*.

[30] Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *FSE'16*, pages 460–473, 2016.

[31] Y. Sui, D. Ye, Y. Su, and J. Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[32] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.

[33] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[34] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171.

[35] Y. Sui, S. Ye, J. Xue, and J. Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience*, 44(12):1485–1510, 2014.

[36] Y. Tang, Q. Gao, and F. Qin. Leaksurvivor: towards safely tolerating memory leaks for garbage-collected languages. In *USENIX Annual Technical Conference*, pages 307–320, 2008.

[37] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA '10*, pages 61–72, 2010.

[38] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *FSE'05*, pages 116–125, 2005.

[39] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. In *PLDI '11*, pages 270–282, 2011.

[40] H. Yan, Y. Sui, S. Chen, and J. Xue. Automated memory leak fixing on value-flow slices for C programs. In *SAC'16*, pages 1386–1393, 2016.

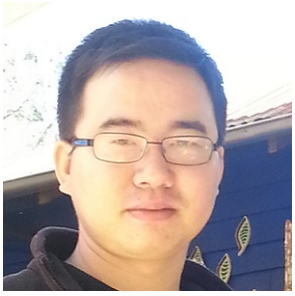
[41] D. Ye, Y. Su, Y. Sui, and J. Xue. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *ISSRE'14*, pages 88–99, 2014.

[42] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined variables with static value-flow analysis. In *CGO '14*, pages 154–164.

[43] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336. Springer, 2014.

ABOUT THE AUTHORS:

Hua Yan is a PhD candidate in the School of Computer Science and Engineering at the University of New South Wales, Australia, supervised Prof. Jingling Xue. He also works part-time at CSIRO, Australia, co-supervised by Dr. Shiping Chen. Before that, he was a software engineer in a commercial bank for 3.5 years. Hua Yan received his B.S. and M.S. in computer science from Peking University, China, in 2007 and 2010, respectively. His research interests are software engineering, software testing, software reuse, and program analysis.



Yulei Sui received the Bachelor's and Master's degrees in computer science from Northwestern Polytechnical University, Xi'an, China, in 2008 and 2011, and the Ph.D. degree in computer science from the University of New South Wales, Sydney, Australia. He has been a Postdoctoral Fellow in Programming Languages and Compilers Group, University of New South Wales, since 2014. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He worked as a Research Intern in Program Analysis Group for Memory Safe C project in Oracle Lab Australia in 2013. Dr. Sui was an Australian IPRS scholarship holder, a keynote speaker at EuroLLVM, and a Best Paper Award winner at CGO'13.

Shiping Chen is a principal research scientist at CSIRO Australia. He also holds an adjunct associate professor title with the University of Sydney through teaching and supervising PhD/Master students. He has been working on distributed systems for over 20 years with focus on performance and security. He has published over 100 research papers in these research areas. He is actively involved in computing research community through publications, journal editorships and conference PC services, including WWW, EDOC, ICSOC and IEEE ICWS/SCC/CLOUD. His current research interests include secure data storage & sharing and secure multi-party collaboration. He is a senior member of the IEEE.

Jingling Xue received the BSc and MSc degrees in computer science and engineering from Tsinghua University in 1984 and 1987, respectively, and the PhD degree in computer science and engineering from Edinburgh University in 1992. He is currently a professor in the School of Computer Science and Engineering, UNSW Australia, where he heads the Programming Languages and Compilers Group. His main research interest has been programming languages and compilers for about 25 years. He is currently supervising a group of postdocs and PhD students on a number of topics including programming and compiler techniques for multi-core processors and embedded systems, concurrent programming models, and program analysis for detecting bugs and security vulnerabilities. He has served in various capacities on the Program Committees of many conferences in his field. He is the senior member of the IEEE.