Path and Context Sensitive Inter-procedural Memory Leak Detection*

Zhongxing Xu
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Graduate University
Chinese Academy of Sciences
xzx@ios.ac.cn

Jian Zhang
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
zj@ios.ac.cn

Abstract

This paper presents a practical path and context sensitive inter-procedural analysis method for detecting memory leaks in C programs. A novel memory object model and function summary system are used. Preliminary experiments show that the method is effective. Several memory leaks have been found in real programs including `which` *and* `wget`.

Keywords: memory leak, path feasibility, bug finding

1 Introduction

Languages like C require the programmer to manually allocate and free memory in programs. Memory leak is a common problem in such programs. Memory leak in long running programs can cause system performance degradation, and even system crash. Many memory leak problems are subtle and hard to find manually. It has few symptoms other than the slow and steady increase in memory consumption.

We developed an automated method to find memory leaks statically. Our method has the following characteristics.

- We use a novel memory object model and escape analysis method. Our modeling of memory objects accommodates both escape analysis and symbolic execution of the program.
- The analysis is path sensitive. We use a constraint solver (CVC3 [3]) to reason about the feasibility of the program path. This can eliminate a large number of false alarms. In addition, path based analysis makes the path that causes the defect immediately available.
- Path sensitive analysis also makes a separate conservative pointer analysis unnecessary, where multiple aliases are maintained. The pointer information is tracked along a single path at a time. Compared with traditional data flow analysis, path based analysis does not merge data flow facts at join points of the CFG. This greatly simplifies data structure and implementation.
- The analysis is context sensitive. Heap objects allocated at different call sites are tracked separately. Function arguments at different call sites are also recognized as different.
- The analysis is inter-procedural. The function summary system is precise and scalable to large programs.

Section 2 gives a simple example to illustrate the power of our analysis. Section 3 gives an overview of the method, and describes the memory objects modeling method. Section 4 describes the data flow lattice used during analysis. Section 5 and 6 give details of the analysis method. Experimental results are shown in Section 7. We discuss related works in Section 8, and conclude in Section 9.

2 An Example

This section presents an example to show the effectiveness of our method. The code in Figure 1 is a simplified version of real world code.

The analysis begins with compiling the code, building the call graph. Then it does its work in function modeling phase and memory leak checking phase.

*This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant number 60633010 and 60673044.

```
int malloc_arg1(int **p) {
  int *t = malloc(8);
  if (!t) return 0;
  else {
    *p = t;
    return 1;
  }
}
int malloc_arg2(int **p) {
  int *t = malloc(8);
  if (!t)  assert(0);
  *p = t;

  if (...)
    return 0;
  else
   return 1;
}

void foo(void) {
  int r, *p;
  r = malloc_arg1(&p);
  if (!r)
L1:    return;
  else {
    // do something to p
    free(p);
  }

  int q = malloc_arg2(&p);
  if (!q)
L2:    return;
  else {
    // do something to p
    free(p);
  }
}
```

Figure 1. A simplified example from real code. In malloc_arg1, the return value 1/0 indicates the success/failure of memory allocation. In malloc_arg2, the return value has no relation to the memory allocation. There is a memory leak when the function foo returns at line L2.

Function	Behavior	Return Value	Feasible Paths
malloc_arg1	MallocArg	1	1
malloc_arg2	MallocArg	Unknown	2
foo	None	void	2

Table 1. The function summary we get after function modeling phase.

During function modeling phase, the functions are visited bottom-up in the call graph, i.e. leaf nodes of the call graph are visited first. The benefit of bottom-up visiting is that the callee's summary information is always available when needed. For each function we limit the loop time to at most once, and generate all feasible paths of it. The detailed path generation and feasibility decision method is given in Section 5. Function modeler visits each feasible path to extract function behavior information.

After the function modeling phase, the function summaries we get are given in Table 1. Function `malloc_arg1()` and `malloc_arg2()` both allocate a memory block and save its address into a variable pointed to by its argument. But `malloc_arg1()` has an associated return value 1 with its memory allocation behavior. In our analysis, we always assume `malloc()` is successful. So there is only 1 feasible path in function `malloc_arg1()`.

Having this precise summary information, the memory leak checker can correctly detect the memory leak at line L2. Since we have no information about the return value of `malloc_arg2()`, the path returning at line L2 is feasible, which makes the heap object malloced in `malloc_arg2()` leaked. On the other hand we get `r = 1` after calling `malloc_arg1()`, which makes the path returning at line L1 infeasible.

3 Overview

The framework of our analysis is shown in Figure 2. The analysis is done in the following steps:

1. The front end compiles and links the program to be checked into a file which contains intermediate representation of all functions of the program. The whole program call graph is built subsequently.
2. Function modeling phase. Visit the functions in the bottom-up order of the call graph. For each function,
   (a) Generate paths for the function.
   (b) Check the feasibility of the paths and save feasible paths.
   (c) Analyze each feasible path to get function summary information.

3. Memory leak checking phase. Visit the functions in any order. For each function, for each of its saved feasible paths, check for possible memory leaks.

The main difficulty of doing inter-procedural analysis is to get a precise description of what a function does at the call site. Our method does the analysis in two phases: function modeling phase and bug detection phase.

In function modeling phase, each function is visited in bottom-up order in the call graph. For each function, we generate all paths from its entry to exit. We traverse each loop at most once to make the number of generated paths limited. Each path is subject to a path feasibility checker before being passed to the function modeler. A path is feasible if there exists a set of input values that can cause the program to execute the path. The feasibility checker is not sound. But it is fast. It does a best-effort decision of path feasibility. Experiments show that it can prune most of the infeasible paths so that these paths will not be passed to later more expensive analysis.

The product of the function modeling phase is a summary of each function. The function summary contains the following information: the feasible paths of the function, the behavior related to heap memory objects, the return value, etc.

In addition to user functions in the program, we also model library functions. A simple function modeling language is used to describe standard C library functions. For example, function `malloc()` can be modeled by

```
malloc { return heapobj },
```

and function `printf` is modeled as ignored:

```
printf { ignored }.
```

Currently there are about 300 functions in the function model library and new functions are added as needed. By limiting the path sensitivity intra-procedurally, we reduce the analysis cost and maintain the high precision.

In the memory leak checking phase, the feasible paths of a function is analyzed for memory leak detection. The analysis focuses on the states of the heap memory objects, not on the pointers that point to them. We model the memory objects in the program explicitly.

3.1 Modeling of Memory Objects

All the memory objects are allocated explicitly in LLVM (our implementation platform, see Section 7). Global objects are defined at the module scope. Stack objects are allocated with `alloca` at the front of a function definition. Heap objects are allocated with `malloc` in the program.

We describe each memory object with the following information.

- *Kind*: A memory object can be **Global**, **Stack**, **Argument**, or **Heap**. In LLVM, the function arguments are passed in virtual register and placed on stack explicitly with the `store` instruction. We identify the object pointed to by a pointer function argument as of kind **Argument**.

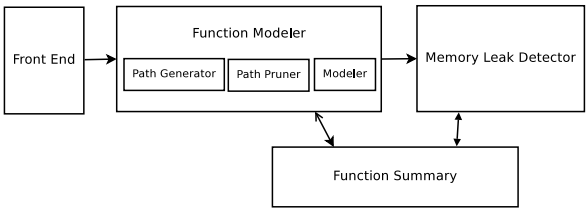

Figure 2. Analysis system overview

- *State*: We associate each heap object with a state: **Malloced**, **Freed**, **Returned**, **EscapedByArg**, **EscapedByGlobal**, **EscapedByUnknown**. Our current analysis algorithm is not field sensitive, i.e. we do not track the pointer fields in a struct or array. We mark the heap object whose address is saved into a struct or array as **EscapedByUnknown**. Due to this, our method can not catch memory leaks caused by recursive data structures.

- *Index*: We abstract the address of a memory object into an integer index. The analysis does not support cross-object address calculation.

- *EscapedBy*: This field records how the heap object is escaped from the function where it is allocated.

- *PointedBy*: This field records the object pointing to this object. It is used in the leak analysis module.

- *SValue*: This is the data flow lattice value of the object used during analysis. The lattice is described in detail in Section 4.

3.2 Escape Model

Given a set of program paths, the memory leak checker tracks the states of heap objects through simulating pointer operations in the path. When we finish analyzing the path, the states of heap objects are examined. If a heap object is neither freed nor escaped, it is recognized as leaked.

Figure 3 shows the escape model: *returned*, *escaped by global variable*, *escaped by argument*, and *escaped by unknown*. The last escape model is subtle. Usually this occurs when the programmer is building recursive data structures. Whenever a heap pointer is stored into a struct field, we mark the heap object as *escaped by unknown*.

Separation of function modeling and memory leak detection makes it easy to extend the system to check other

```
void *foo()
{
  void *p = malloc(8);
  return p;
}
```

The heap object is *returned*.

```
void *p;
void foo()
{
  p = malloc(8);
}
```

The heap object is *escaped by global variable* `p`.

```
void foo(void **p)
{
  *p = malloc(10);
}
```

The heap object is *escaped by argument* `p`.

```
struct list *head;
void foo(void) {
  struct list *n =
         malloc(sizeof(struct list));
  head->next = n;
}
```

The heap object n is *escaped by unknown*.

Figure 3. The escape model used by the analysis.

kinds of program bugs. We can extend the modeler to extract more function properties, and plug other checkers into the system.

4 Dataflow Value Lattice

We designed a lattice for our analysis, as shown in Figure 4. There are 3 types of values in the lattice: Boolean, integer, and pointer.

- Boolean:
  - TRUE.
  - FALSE.
  - SymbolicBool. A symbolic Boolean value is a predicate at branches, such as `x > 3`.
- Integer:
  - Concrete Integer. For example, 3, 5, 8, ...
  - Symbolic Integer. A symbolic integer is an expression of integer type, which may be constructed from statements like `x += y`, where `x` or `y` has no concrete integer value.
- Pointer:
  - Null.
  - NonNull. A NonNull pointer is definitely not NULL, but we know nothing more about it.
  - ConcretePointer. A concrete pointer takes the index of the memory object it points to as its value.
  - SymbolicPointer. A symbolic pointer is an unknown pointer value, e.g., `p = strchr(s, '.');` `p` gets a symbolic pointer value pointing at somewhere of string `s`.

This lattice can retain much of the information of the program. The uncertain part (i.e. symbolic values) leaves space for the constraint solver. We send these expressions to the constraint solver and query it for feasibility of the path.

5 Function Modeling

The purpose of function modeling is to get two kinds of information of a function: the set of feasible paths, which are to be analyzed by later memory leak detection phase, and the function's behavior on heap memory objects.

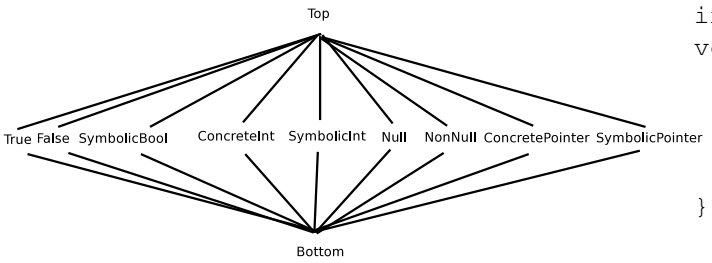

Figure 4. The lattice used in data flow analysis.

5.1 Function Summary

What a function might do about a heap memory object? We summarize the function behavior into the following categories:

- *None*: The function does nothing relevant to heap objects.
- *ReturnHeapObj*: The function allocates a heap object, and returns its address.
- *MallocGlobal*: The function allocates a heap object, and saves its address into a global variable.
- *MallocArg*: The function allocates a heap object, and saves its address into a variable pointed to by an argument of the function.
- *FreeGlobal*: The function frees a heap object pointed to by a global variable.
- *FreeArg*: The function frees a heap object pointed to by an argument of the function.

After initial experiments, we have observed some code patterns occur frequently. It is necessary to extend the original function summary by adding condition to the behavior.

```
int foo(int **p) {
  int *t = malloc(8);
  if (t) {
    *p = t; return 1;
  } else {
    return 0;
  }
}
```

Figure 5. return value is related to malloc

In Figure 5, the function's return value indicates the result of memory allocation. `foo()` returns 1 when memory

```
int *g;
void bar() {
  if (!g)
    g = malloc(8);
  else
    // do not allocate
}
```

Figure 6. the precondition of malloc is g == 0

allocation succeeds, and returns 0 when memory allocation fails. In Figure 6, global variable `g` gets malloced only when it has not been malloced.

A common property of the code fragments in Figure 5 and Figure 6 is that they associate a condition with the memory allocating behavior. Without modeling such property, we get a lot of false alarms. But not all of the return values or path conditions are associated with a memory allocating behavior. In the contrived example of Figure 7, the return value has nothing to do with the `malloc()` at line 3.

```
int foo(int **p, int x) {
  *p = malloc(8);
  if (x > 3)
    return 1;
  else
    return 0;
}
```

Figure 7. return value is not relevant to malloc

To establish correct association between function behavior and return value, we collect all the return values of paths which have memory allocating behavior and do a meet operation of the lattice in Section 4 on them. For a malloced global variable, we only record the path condition involving that variable.

The refined function summary information is:

- *None*: The function does nothing relevant to heap objects.
- *ReturnHeapObj*: The function allocates a heap object, and returns its address.
- *MallocGlobal*: The function allocates a heap object, and saves its address into a global variable.
- *MallocGlobalCond*: The same as above, but has an associated condition.
- *MallocArg*: The function allocates a heap object, and saves its address into a variable pointed to by an argument of the function.

```
Model -> Name { Behavior }
Name -> [A-Za-z0-9_]*
Behavior -> ignored
          | return Value
Value -> heapobj
        | int
        | pointer
        | @1
```

Figure 8. The language for modeling library functions.

Model	Semantic Meaning
`random { return int }`	return a symbolic integer.
`strchr { return pointer }`	return a symbolic pointer.
`strcat { return @1 }`	return the first argument.

Table 2. Some examples of library function modeling.

- *MallocArgRetVal*: The same as above, but has an associated return value.
- *FreeGlobal*: The function frees a heap object pointed to by a global variable.
- *FreeArg*: The function frees a heap object pointed to by an argument of the function.

5.2 Library Function Modeling

The C programming language has a comprehensive standard library. Ignoring these library functions may result in imprecise analysis of programs. We designed a function modeling language to alleviate the labor of modeling these functions manually. Currently the language is memory leak detection biased. But it can be extended easily.

The syntax of the modeling language is given in Figure 8. Table 2 shows some examples of library function modeling.

Currently there are about 300 functions in the function model base. New functions are added and the modeling language are extended as needed.

5.3 Path Generation

The function modeling is path sensitive. A path generator is run on the visited function. The path generator traverses the CFG of the function in the depth-first order, and generates all paths from the entry to the exit of the function. Loops are identified by an earlier loop analysis pass, and are traversed at most once. This guarantees the termination of the traversal. Experiments show that the zero/once execution of the loops is effective for memory leak detection.

Doing the analysis on a path basis has several advantages.

- Analyzing a single path at a time simplifies pointer alias analysis greatly. It is no longer necessary to do a conservative alias analysis. A pointer can point to at most one position at any time during the execution of the path. Usually this position can be known precisely.
- It helps programmer to locate the bug. Once a memory leak is identified, the path that causes it is available immediately to check manually. This advantage greatly helps us during the implementation of the method.
- It helps reduce the false alarms. With the help of a high quality path feasibility checker, we can eliminate most of the false alarms.

The disadvantage of path based analysis is relative higher cost, since the number of paths grows exponentially with the branches in the CFG. But as we limit the loop traversal to at most once, the number of paths generated for most functions is less than 1000.

A generated path is a list of basic blocks. In later analysis phases we update the information associated with virtual registers and memory objects during the checking. In addition, we record the indices of heap objects related to a path.

5.4 Path Pruner

We use symbolic execution on the lattice of Figure 4 to decide the feasibility of a path. At the beginning of analysis of a path, the global variables and function arguments are set to symbolic values. The symbolic execution is conducted faithfully according to the operations of the program path.

Some operations that we currently do not model are ignored, and their values are set to the bottom in the lattice. Such operations include floating-point operations, division, bitwise operations, etc. Function calls are modeled according to function summary information obtained during function modeling phase. Since we traverse the call graph in the bottom-up order, usually we can have the function summary when a function is called. Note that we do not apply function summary fully in path pruner, since the function summary is memory leak checking biased, and may not be suitable for general purpose path pruning. When recursive function call is encountered, and the summary of the callee is not available at the time, we do not apply the function summary.

When a path finishes execution, the path condition at each branch is collected and solved by the constraint solver. Theoretically a constraint solver (or theorem prover) is too

expensive for program analysis. But as we observed in the experiments, almost all of the constraints are very easy to solve. The constraint solver does not turn out to be too heavy as traditionally viewed.

The feasible paths are saved in the function summary, and are passed to function modeler and memory leak checker.

5.5 Function Modeler

Function modeler extracts the heap behavior of the function as described in Section 5.1. The modeler analyzes each feasible path obtained in Section 5.4. The modeler focuses on pointer values and heap object states.

When a function call is encountered, the modeler references the function summary to see if it has behavior related to heap objects, whether it is malloc or free, and applies the behavior accordingly.

When a heap pointer is saved to some variable, the modeler changes the state of the heap object according to the storage class of the destination. If it is a global variable, the modeler sets the state of the heap object to `EscapedByGlobal` and records the global variable. If it is an argument, the modeler sets the state to `EscapedByArg`. If it is a local variable, do nothing. If it is a struct field, the modeler sets the state of the heap object to `EscapedByUnknown`, because most of the time such behavior implies the programmer is building recursive data structure. The old heap object pointed to by the destination variable, if any, should be set to `Malloced` state. If a heap pointer is returned, the state of the heap object is set to `Returned`.

When it finishes analyzing the path, the modeler checks the state of each heap object. If there is an `EscapedByArgument` object, the modeler records the index of the argument and the related return value in the function summary. If there is an `EscapedByGlobal` object, the modeler records the associated path condition involving the global pointer. If there is a `Returned` object, the modeler records the function as `ReturnHeapObj`.

6 Memory Leak Checker

Once we have a model of all the functions in the program, we do the actual memory leak analysis. Memory leaks caused by recursive data structure manipulation can not be detected, because that needs global pointer analysis which is extremely difficult for C.

We still visit the functions in the bottom up order in the call graph, although this is not required. For each function, we analyze all feasible paths saved in the function summary.

The path pruner focuses on path condition extraction, while the function modeler focuses on heap object state updates. The memory leak checker collects both information. Why is this necessary? Because the integration of memory object information and path predicates information may result in new infeasible paths which can not be detected by the path pruner.

Consider again the example in Figure 1. The path in function `foo()` that returns at line `L1` is not recognized as infeasible by the path pruner. Since the path pruner does not simulate function behavior fully according to the function summary. The path is really feasible when memory allocation fails. In memory leak checker it is viewed as infeasible because we assume every memory allocation is successful. For checking other types of bugs, such assumption may not be appropriate.

So in memory leak checker we collect as much information as we can, and simulate function behavior fully according to the function summary.

When the checker finishes analyzing a feasible path, the state of each heap object is examined. For every heap object whose state is `Malloced`, we query the constraint solver for the feasibility of the path. If the path is feasible, the checker emits a warning.

7 Implementation and Experiments

We have implemented the analysis in the LLVM compiler infrastructure [6]. LLVM is a low-level object code representation that uses simple RISC-like instructions, but provides rich, language-independent, type information and data flow (SSA) information about operands. Memory objects are allocated explicitly in LLVM. All computation occurs in virtual registers. Virtual registers and memory locations are distinct. It is not possible to take the address of a virtual register, and virtual register can only represent scalar variables. Values are communicated between virtual registers and memory objects with explicit load and store instructions.

We used numerous LLVM facilities including a C front end, call graph constructor, loop analysis, and others.

We use the existing building mechanism of the program (usually GNU autotools) to compile the whole program into a big bitcode file of LLVM. This bitcode file contains all the code of the program. Full inter-procedural analysis can be done on this file.

We have run our system on code used daily by people, including `make`, `wget`, `bzip2`, `gzip`, `adns`, `time`, `proftpd`, `which`, etc. The sizes of these packages are between 2,000 to 50,000 LOC. All experiments are carried out on a PC with a Core2 Duo T7600 (2.33GHz) CPU, 2GB memory. The checking time is under 5 minutes for each package.

Two of the memory leaks found by the system are shown in Figure 9 and Figure 10. We reported to the developers of

the programs, and they confirmed the bugs. Both of the bugs can only be found with the presence of inter-procedural data flow information. The control flow of the code is complex, and is hard to analyze manually.

```
// in file which.c:
int path_search(...) {
  if (...) {
    ...
    do {
M:    result = find_command_in_path(...);
      if (result) {
        ...
        if (...) {
        } else if (in_home) {
          if (skip_tilde) {
            next = 1;
L:          continue;
          }
          ...
        }
        ...
      }
      free(result);
    } while (...)
  }
}
```

Figure 9. Memory leak in which 2.16: result gets allocated a heap object at line M. If the execution of program reaches line L, the statement free(result) will be skipped, and result will get allocated another heap object. The old one is leaked.

```
// in file ftp-basic.c:
uerr_t ftp_pasv(...) {
   ...
M: err = ftp_response (csock, &respline);
   ...
   s = respline;
   for (s += 4; *s && !ISDIGIT (*s); s++);
   if (!*s)
L:   return FTPINVPASV;
   ...
}
```

Figure 10. Memory leak in wget 1.10.2: ftp_response() allocates a heap object and saves the address to the position pointed to by its second argument. So respline is allocated a heap object at line M, but is not freed when function returns at line L.

8 Related Works

Memory leak detection using dynamic techniques has been a part of the programmer's toolkit for more than a decade. Purify [4] and Valgrind [8] are two representative dynamic tools. Dynamic memory leak detection is limited by the quality of the test suite; unless a test case triggers the memory leak it cannot be found. We restrict the discussion to static techniques for detecting memory leaks. A famous static analyzer is Prefix [1] which can analyze large programs efficiently. However, it does not have precise path feasibility checking.

Saturn [9] is the most similar work to ours. It reduces the problem of memory leak detection to a Boolean satisfiability problem, and then uses a SAT solver to identify potential errors. Their escape analysis is similar to ours: any allocated location in a procedure that is not deallocated nor escaped is leaked. But we use a different function summary system and different path feasibility decision method. Specifically, our function summary captures escapes caused by pointers passed out by function parameters. Their function summary captures escapes caused by pointers passed in by function parameters. So Saturn can not detect the memory leak in Figure 1 and Figure 10. In addition, we use a different approach to do program analysis. Saturn reduces all program operations into Boolean formulas. We analyze the program with traditional data flow analysis and symbolic execution, and use an SMT solver. Our approach is more precise and efficient, because not all of the C program semantics can be naturally translated into Boolean satisfiability problems.

Clouseau uses an ownership model to track an object's owning reference [5]. In this model, every object is pointed to by one and only one owning pointer. They assume that the pointer member fields in an object either always or never own their pointees at public method boundaries, and the destructor method of an object contains code to delete all and only objects pointed to by owning member fields. This idealized ownership model is best suited for well designed object-oriented C++ programs. It can easily be violated by versatile C programs.

LC [7] runs a backward heap analysis to disprove the presence of memory leak. To determine whether a memory leak can occur at a program point, the analysis uses a reverse inter-procedural flow analysis to disprove its negation.

Fastcheck [2] detects memory leak via guarded value flows. It tracks the flow of values from allocation points to deallocation points using a sparse representation of the pro-

gram consisting of a value flow graph that captures def-use relations and value flows via program assignments.

Neither LC nor Fastcheck models functions as sophisticated as our analysis does. They consider path feasibility to a limited extent, i.e., using a SAT solver, which is insufficient to prune false alarms.

We applied LC to the example in Figure 1, and LC reported two warnings for it. It did not eliminate the false alarm at line L1. In constrast, our tool reports only one warning at line L2. (We failed to install and run fastcheck due to lack of instructions on its website. So we were not able to compare it with our tool.)

9 Conclusion

We have presented an automated static memory leak analysis method. The method uses a novel program memory model and inter-procedural function summary to do path and context sensitive analysis. The analysis reasons about program behavior on almost all paths and solves the path condition with a constraint solver. The omited paths are mostly ones with loops in them. Several memory leaks have been found in real programs. Besides memory leak, the method can be easily extended to check for double free and null pointer dereferences.

References

[1] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30:775–802, 2000.

[2] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[3] CVC3. http://www.cs.nyu.edu/cvc3.

[4] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[5] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the 2003 ACM SIGPLAN conference on Programming language design and implementation*, 2003.

[6] The LLVM compiler infrastructure. http://llvm.org/.

[7] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Proceedings of the International Static Analysis Symposium (SAS '06)*, 2006.

[8] Valgrind. http://www.valgrind.org.

[9] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of ECSE/FSE 2005*, 2005.