2011 18th Asia-Pacific Software Engineering Conference

Memory Leak Detection Based On Memory State Transition Graph

Zhenbo Xu
Department of Computer Science and Technology
University of Science and Technology of China
Email: xuzb@ios.ac.cn

Jian Zhang, Zhongxing Xu
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Email: {zj, xzx}@ios.ac.cn

***Abstract*—Memory leak is a common type of defect that is hard to detect manually. Existing memory leak detection tools suffer from lack of precise interprocedural alias and path conditions. To address this problem, we present a static interprocedural analysis algorithm, which captures memory actions and path conditions precisely, to detect memory leak in C programs. Our algorithm uses path-sensitive symbolic execution to track the memory actions in different program paths guarded by path conditions. A novel analysis model called *Memory State Transition Graph* (MSTG) is proposed to describe the tracking process and its results. An MSTG is generated from a procedure. Nodes in an MSTG contain states of memory objects which record the function behaviors precisely. Edges in an MSTG are annotated with path conditions collected by symbolic execution. The path conditions are checked for satisfiability to reduce the number of false alarms and the path explosion. In order to do interprocedural analysis, our algorithm generates a summary for each procedure from the MSTG and applies the summary at the procedure's call sites. Our implemented tool has found several memory leak bugs in some open source programs and detected more bugs than other tools in some programs from the SPEC2000 benchmarks. In some cases, our tool produces many false positives, but most of them are caused by the same code patterns which are easy to check.**

***Keywords*-memory leak; bug finding; static analysis; symbolic execution;**

I. Introduction

Memory leak is a common type of defect in software systems written in languages with explicit memory management like C or C++. It occurs when dynamically allocated memory has never been freed, which consumes the available memory of the system and degenerates the system performance. Eventually, the available memory may be exhausted and causes the system to crash.

This paper presents a static interprocedural analysis algorithm based on Memory State Transition Graph (MSTG) to detect memory leak in C programs. The following list describes the main features of our algorithm.

- **Concise memory abstraction** is used to model memory objects related to memory leak as two kinds, i.e. heap objects and external objects. The heap objects that denote the memory allocated in the heap space are checked for the existence of leaked state and the external objects that represent the unknown memory inside a procedure (such as a memory object passed in by a global pointer) provide alias information during interprocedural analysis. Generally, "memory objects" in this paper refer to heap objects or external objects.
- **A novel analysis model** called *Memory State Transition Graph* (MSTG) is proposed to analyze the C programs. Each MSTG is generated from a procedure by using path-sensitive symbolic execution. The states of memory objects contained in MSTG's nodes indicate whether leaked states exist in the procedure. Memory actions as supplemental data of states can capture the function behaviors precisely. Predicates, namely path conditions annotated on MSTG's edges, are checked for satisfiability to reduce the number of false alarms and the path explosion.
- **Precise procedure summaries** generated from MSTGs store the memory actions and the path conditions. When applying the summaries, the memory actions are used to capture procedure's side-effects and the path conditions are checked for satisfiability to prune the infeasible memory actions. It also helps to reduce the number of false alarms and the path explosion.

We have applied our tool to some GNU open source projects. Five bugs were found in *wget* and one bug in *which*. Compared with a summary-based path-sensitive memory leak detector [1], our tool found more bugs. We have also applied it to analyze some programs in SPEC2000 benchmarks. The experimental results show that some extra bugs were found by our tool while other tools [2]–[4] missed them.

The remainder of the paper is organized as follows. Section II gives a motivating example to show the effectiveness of our memory leak detector. Section III describes our algorithm framework. Section IV introduces our intraprocedural analysis model including memory abstraction and memory state transition graph. Section V describes the interprocedural analysis including summary generation and summary application. Limitations are discussed in Section VI. Experimental results are shown in Section VII. Related work is discussed in Section VIII, and the conclusion and future works are given in Section IX.

1530-1362/11 $26.00 © 2011 IEEE
DOI 10.1109/APSEC.2011.22

II. Motivation and Examples

This section will use a motivating example to illustrate the weaknesses of existing tools in Section II-A and the effectiveness of our memory leak detector in Section II-B.

A. Weaknesses of Existing Tools

The ability of existing tools are mainly limited by path-insensitivity and imprecise summaries of function behaviors. The code fragment in Figure 1, which is used to describe the necessity of recording precise memory actions and path conditions, is a simplified version of some real world programs such as Samba and the Linux kernel. The `bufSelect` function in this figure manipulates a selection between a smaller but faster buffer and a larger but slower heap buffer. The selected buffer is passed out by the second parameter `int **p`. Function `foo1` and `foo2` call `bufSelect` to get a buffer.

```
 1  void bufSelect(int len, int **p,
 2                 int *fastbuf) {
 3    if (len <= 10) *p = fastbuf;
 4    else *p = (int *)malloc(len);
 5  }
 6  void foo1() {
 7    int len = 16;
 8    int *p, fastbuf[10];
 9    bufSelect(len, &p, fastbuf);
10   ... //use p
11  }
12  void foo2() {
13    int len = 8;
14    int *p, fastbuf[10];
15    bufSelect(len, &p, fastbuf);
16    ... //use p
17  }
```

Figure 1. A simple example with memory leak

Consider the function `foo1`, since the value of the variable `len` is greater than 10, a heap object is allocated to `*p` at the call site at line 9. A memory leak occurs at the exit of the procedure when the heap object is not released. Saturn [5] and FastCheck [3] miss this error because they cannot handle the function behavior that heap objects are escaped by parameters.

Function `foo2` doesn't have memory leak while some tools make a mistake. The call site at line 15 just assigns `fastbuf` to `*p` and no heap memory is allocated. As the function `bufSelect` contains two program paths, one of which allocates a heap object and the other one doesn't, the tools without intraprocedural path-sensitivity [2], [4], [6] merge these two paths into an allocation path which means the call site at line 15 is analyzed as an allocation site. Thus false alarms will be produced by these tools. The tool [1] without interprocedural path-sensitivity, that is merging all of the memory behaviors in different paths at the exit of functions, also summarizes the function `bufSelect` as an allocation one and produces false alarms.

B. Leak Detection using Memory State Transition Graph

In the above example, some tools miss errors, due to the lack of global pointer analysis and precise function behavior model. Others report false positive because the path conditions are discarded during intra- or inter-procedural analysis. With a precise Memory State Transition Graph model, our tool can identify the memory leak in Figure 1. Besides, the false positive can be eliminated.

Firstly, we analyze the function `bufSelect` as an MSTG shown in Figure 2. The operator `Reg(p)` represents the memory object the pointer `p` points to. At the beginning, the external objects `Reg(p)`, `Reg(*p)`, `Reg(fastbuf)` passed in by function parameters are initialized to `Accessed` state and their action sets are empty. Then we compute the satisfiability of the first branch condition `len<=10`. If it is satisfiable, the condition will be annotated on the edge, the action assigning `Reg(fastbuf)` to pointer `*p` will be recorded in the action set, and the state of `Reg(fastbuf)` will turn to *Escaped*. At the same time the `Reg(*p)` is removed as no pointer points to it. Another branch with condition `!(len<=10)` allocates a new heap object to pointer `*p`. The action set also records the changes. If leaked states exist in nodes, leak bugs will be reported.

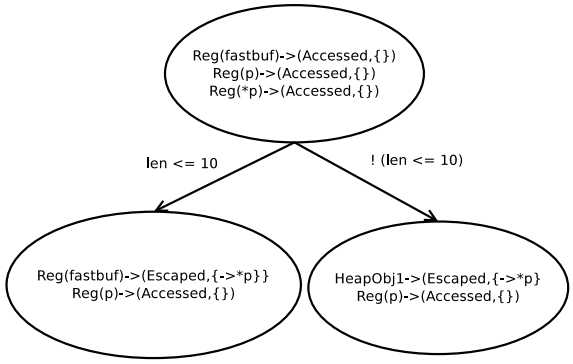

Figure 2. The Memory State Transition Graph of the function $bufSelect$

After finishing the analysis of the function `bufSelect`, we generate the function summary for `bufSelect` from the MSTG. The summary consists of the annotated path conditions and the action sets at the leaf nodes. The following shows the summary of `bufSelect`.

$$\{P : len <= 10, UActs : Reg(fastbuf) \rightarrow *p\} \bigcup$$
$$\{P :!(len <= 10), UActs : HeapObj1 \rightarrow *p\}$$

where P is a set of predicates that denotes the path conditions and $UActs$ is a union set of actions in leaf nodes. The summary is applied at the function's call site. At line 9, the value of the variable `len` is greater than 10, so the

function `bufSelect` allocates a heap object to the pointer `*p`. If the heap object is not freed at the end of function `foo1`, it will cause a memory leak. At line 15, no memory leak will be reported because the second branch condition `!(len<=10)` is unsatisfiable. As the MSTG (summary) records precise memory actions and path conditions, our tool can detect the memory leak in function `foo1` and eliminate the false alarm in function `foo2`.

III. Algorithm Framework

Figure 3 is the framework of our algorithm. The $LeakDetect$ algorithm first builds a call graph from the whole program C and then visits each function in a bottom-up order in the call graph to assure a called function has been analyzed. The $Visit$ algorithm generates an MSTG for function f and reports the detected bugs during the generation. Finally, the summary Sum_f of function f is generated from the built MSTG. The $GenMSTG$ algorithm and $GenSum$ algorithm in $Visit$ will be described in the following sections.

input: the whole program C
output: leak reports
$LeakDetect$(program C)
1: Build Call Graph CG from C
2: **for all** function f in bottom-up order in CG **do**
3: $Visit(f)$
4: **end for**
$Visit$(function f)
1: $MSTG_f$ = $GenMSTG(f)$
2: **if** leak bugs exist **then**
3: output leak reports
4: **end if**
5: Sum_f = $GenSum(MSTG_f)$

Figure 3. Detection Algorithm

IV. Intraprocedural Analysis

This section describes the memory abstraction, basic definitions of MSTG and intraprocedural analysis by using MSTG.

A. Memory Abstraction

All of the memory regions except allocated in stack are abstracted into heap objects and external objects. Heap objects are the memory objects that are dynamically allocated by library functions such as $malloc$, $realloc$, etc. We use the set $\{Heaps\}$ to represent heap objects. External objects are the memory objects passed in by pointers of function parameters and global variables, represented with the set $\{Externs\}$. Another set $\{Statics\}$ describes the variables stored in static space such as global variables, static variables. Note that $\{Externs\}$ and $\{Statics\}$ are disjoint.

Suppose $m \in \{Heaps\} \cup \{Externs\}$ where m is the memory object to be abstracted.
ToExtern (that contains **ToArg** and **ToGlobal**): $m \rightarrow p$ where $Base(p) \in \{Externs\} \cup \{Statics\}$
ToAlloc: $m \rightarrow p$ where $Base(p) \in \{Heaps\}$
ToReturn: $m \rightarrow rv$ **ToFree**: $m \rightarrow free$

Figure 4. The notations of the actions for each memory object

For example, given a global variable `int *g`, `g` belongs to $\{Statics\}$ while `Reg(g)` belongs to $\{Externs\}$ (Suppose that the memory region `g` points to is unknown). We use the notation $Base(p)$ to represent the memory region where the pointer p is stored in.

According to the above memory abstraction, we define the following 4 actions for each memory object and the notations are shown in Figure 4.

- *ToExtern* action that assigns memory objects to the pointers whose base regions belong to $\{Externs\}$ or $\{Statics\}$. It contains *ToArg* action and *ToGlobal* action. As shown in Figure 4, m is the memory object to be abstracted and is assigned to a pointer p of which $Base(p)$ belongs to $\{Externs\}$ or $\{Statics\}$.
- *ToReturn* action that returns a memory object. We use rv to represent the return value in Figure 4.
- *ToAlloc* action that assigns memory objects to pointers whose base regions are heap objects.
- *ToFree* action that frees a memory object. The symbol $free$ in Figure 4 is an abstract value which denotes m is freed.

The states of heap objects and external objects will be affected by these actions. We define the following 5 memory states for each heap object.

- *Allocated* state denotes heap objects are initially allocated.
- *Escaped* state denotes heap objects are assigned to argument pointers or global pointers or heap objects are returned (including *ToExtern* and *ToReturn* actions).
- *Freed* state denotes heap objects are freed (including *ToFree* action).
- *Relinquished* state denotes heap objects are assigned to some complex C expressions that we cannot handle.
- *Leaked* state denotes heap objects are not freed, escaped or relinquished at the end of the function.

We also present 4 memory states for each external object.

- *Accessed* state denotes external objects are initialized.
- *Escaped* state denotes external objects are assigned to argument pointers or global pointers, or external objects are returned (including *ToExtern* and *ToReturn* actions).
- *Freed* state denotes external objects are freed (including *ToFree* action).

- *Relinquished* state denotes external objects are assigned to some complex C expressions that we cannot handle.

These states will be used in the following sections. Note that although the *ToAlloc* action is defined, we treat a memory object with this action as a relinquished one. That means data structures like list, queue, etc cannot be reasoned about.

B. Memory State Transition Graph

1) Basic Definitions: Based on the memory abstraction above, we define a Basic State Transition Model (BSTM) for a single memory object. A BSTM is a tuple $\langle S, P, \rightarrow \rangle$, where

- S is a set of memory states. It consists of the states of heap objects and external objects. *Allocated* and *Accessed* are two initial states. The others can be treated as terminal states which represent a memory object's state at the exit of the procedure.
- P is a set of predicates. Each predicate in P represents a set of symbolic conditional expressions extracted from the procedure's path conditions using symbolic execution.
- $\rightarrow \subset S \times P \times S$ is a ternary relation of labelled transitions. The definition of $\rightarrow$ is a tuple $\langle cs, p, ns \rangle$ where cs and ns respectively denote the current state and the next state of the transition. It can be written as $cs \xrightarrow{p} ns$ which represents the transition from state cs to state ns guarded by predicate p.

We respectively define the BSTM for a single heap object and external object in Figure 5 and Figure 6. These two figures describe the transition relations among states.

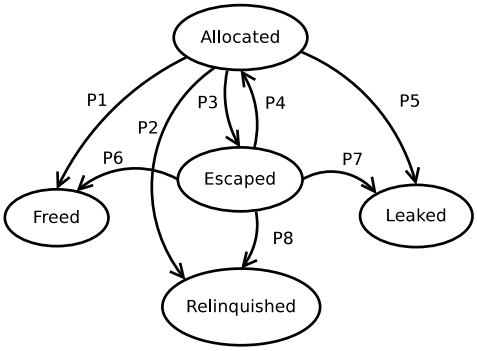

Figure 5. Basic State Transition Model for heap objects

Since a state transition of a memory object may affect the states of other memory objects, the BSTM is not enough to track the behaviors of a program precisely. In order to capture the relations among different memory objects, we define a Memory State Transition Graph as a tuple $\langle S, M, P, \rightarrow \rangle$, where

- S is a set of memory states. Each state is composed of a *Kind* attribute and a set of memory actions.

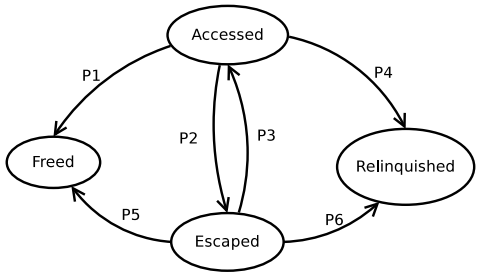

Figure 6. Basic State Transition Model for external objects

- M is a set of memory objects.
- P is a set of predicates same as the definition in BSTM.
- $\rightarrow \subset (M \times S) \times P \times (M \times S)$ is a ternary relation of labelled transitions.

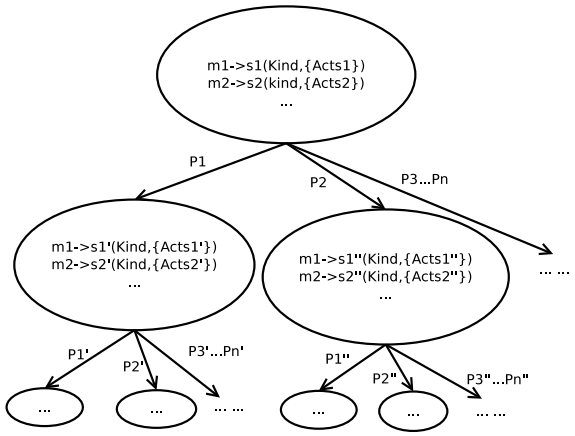

Figure 7. Memory State Transition Graph

The Memory State Transition Graph is shown in Figure 7. Each node contains a set of memory objects and their states. Each state has a $\langle Kind, \{Acts\} \rangle$ pair. Edges are annotated with predicates. The $Kind$ attribute is actually equal to the state definition in BSTM. For convenience, we still call it "state" in MSTG. The $Acts$ describe the memory actions that cause the memory object to transit to the state.

2) Memory State Transition Graph Generation: We use path-sensitive symbolic execution to construct memory state transition graphs from a program. Figure 8 describes the generation algorithm. We first generate the control flow graph (CFG) for function f, and use the breadth-first search (BFS) algorithm to traverse the CFG. The MSTG is generated during the traversal.

When analyzing each block, we are concerned about assignment statements from pointers to pointers, free statements, return statements, call statements and branch statements. The three former types of statements cause a states and actions update (the return value as a predicate is added to current path conditions). If leaked states exist during the update, bugs will be added to the bug reporter. Summaries

are applied at the call statements and the branch statements generate two new child nodes. We don't list the handling of switch statements in the algorithm for clearly. The algorithm $ApplySum$ will be given in the next section. The generation of MSTG is actually intraprocedural detection.

$GenMSTG$(function f)
1: Generate control flow graph CFG for f
2: Get root block $RootB$ from CFG
3: Init an empty memory state transition graph $MSTG_f$
4: Generate root node $RootN$ for $MSTG_f$
5: $BFS(RootB, RootN)$
6: **return** $MSTG_f$

BFS(Block B, Node N)
1: **for all** stmt (statement) s in B **do**
2: **if** s is a pointer assignment stmt, return stmt or free stmt **then**
3: update states and actions in N
4: **else if** s is call stmt, cf is the callee function **then**
5: $ApplySum(Sum_{cf}, N)$
6: **else if** s is branch stmt with condition C **then**
7: **if** C is satisfiable **then**
8: LN=NewNode(N, C), BFS(LB, LN)
9: **end if**
10: **if** !C is satisfiable **then**
11: RN=NewNode(N, !C), BFS(RB, RN)
12: **end if**
13: **else**
14: do symbolic execution
15: **end if**
16: **end for**

Figure 8. The algorithm of MSTG generation

V. Interprocedural Analysis

This section describes the summary-based approach to interprocedural leak detection. Section V-A defines the summary representation and discusses how to generate a summary from an MSTG. Section V-B shows summary application at function's call sites.

A. Summary Generation from MSTG

After MSTG has been generated, intraprocedural analysis is finished. In order to avoid analyzing the same procedure twice, we use the existing MSTGs to generate a summary for each procedure.

The following definition describes the summary representation.

$$\{P_1, UActs_1\} \bigcup \{P_2, UActs_2\} \bigcup ... \bigcup \{P_n, UActs_n\}$$

where P_i denotes the predicate and $UActs_i$ denotes a union set of the $Acts$ in MSTG. A summary is composed of several $\langle P, UActs\rangle$ pairs. P in a $\langle P, UActs\rangle$ pair represents a conjunction of path conditions from the start node to a leaf node. The $UActs$ is extracted from the states in leaf nodes. In leak analysis, the side-effects we are interested in are whether the function allocates new heap objects and which external object is assigned to external pointers. Actions in the leaf nodes are retained according to the following rules:

1) If the state of a heap object is *Freed* or *Relinquished*, the actions referring to this object will be discarded.
2) If the state of a heap object is *Leaked*, the actions referring to this object will be discarded. It will report a memory leak during intraprocedural analysis.
3) If the state of a heap object is *Escaped*, the actions *ToExtern* and *ToReturn* will be retained.
4) If the state of an external object is *Freed* or *Relinquished*, the *ToFree* action will be retained. We treat a *Relinquished* object as a *Freed* one.
5) If the state of an external object is *Escaped*, the actions *ToExtern* and *ToReturn* will be retained.

After applying these rules, the retained actions are stored as an $UActs$ set. Figure 9 shows the algorithm of summary generation.

$GenSum(MSTG)$
1: Generate an empty summary Sum
2: **for all** leaf node N in $MSTG$ **do**
3: Get the conjunction of conditions P in the path from start node to N
4: Generate memory actions $UActs$ according to rule $1 \sim 5$
5: Store $\langle P, UActs\rangle$ to Sum
6: **end for**
7: **return** Sum

Figure 9. The algorithm of summary generation

B. Summary Application

A summary is a union of several $\langle P, UActs\rangle$ pairs. Figure 10 introduces the algorithm of summary application and the following steps describe the details.

$ApplySum$(Sum, N)
1: Map actual parameters to formal parameters
2: **for all** $\langle P, UActs\rangle$ in Sum **do**
3: **if** P is satisfiable **then**
4: Apply $UActs$ to N
5: **end if**
6: **end for**

Figure 10. The algorithm of summary application

1) Map the actual parameters to the formal parameters by using access paths. Access paths were first used by Landi and Ryder [7] as symbolic names for memory locations accessed in a procedure.

2) Check for the satisfiability of P. Since many symbols in P may come from function parameters, after step 1, the satisfiability is easier to reason about and the feasible parts in summaries are reduced.
3) Apply the actions $UActs$ to the caller's context. The three components of actions we should process are external objects, heap objects and pointers whose base region is external or heap object. External objects should be mapped to the actual memory object in the caller by using access paths. Heap objects are treated as the caller allocates new heap objects. Since external objects and heap objects are processed, we just replace pointers' base region with the actual objects. Then the actions can be easily applied to the caller. Take the following code as an example.

```
void myfree(int *x) { free(x);}
void f() {
  int *m = malloc(32);
  myfree(m); }
```

The action of the function `myfree` recorded in summary is `ToFree(Reg(x))`. When `myfree` is called, we first map the `Reg(x)` to the heap object the pointer `m` points to, and then apply *ToFree* action to the heap object.

VI. Limitations

The limitations of our approach are mainly caused by imprecise handling of loops, recursions and recursive data structures.

- *Loops*: We bound the number of loop iterations. Fewer paths are analyzed and the summaries of these paths are absent. That means we may miss some bugs in these paths. But when analyzing the bugs in real programs, we found that memory leaks in loops usually occur since the second iteration. Leak bugs found at the third or later iterations are caused by the same statements as the second iteration. Generally, leak bugs in loops can be detected without a deeper iteration.
- *Recursions*: If two functions are strongly connected in call graph or a function calls itself, recursions exist. We perform a conservative analysis to the recursive functions. That is, the variables in the callee's side-effects are assigned an unknown or symbolic value.
- *Recursive data structures*: Since a heap object with the action *ToAlloc* is treated as a relinquished one, the heap objects allocated in data structures like list, queue, etc cannot be detected.

VII. Implementation and Experiments

We have implemented our tool called $Melton$ in Clang [8], a new C family front-end for the LLVM compiler. Clang provides a powerful intraprocedural framework for static analysis. We extend it to support interprocedural analysis for memory leak detection. The satisfiability checking is answered using the solver in Clang. The following sections describe our experiments.

A. Leak Errors Found in Open Source Projects

Melton was applied to some GNU open source projects, and it found 5 memory leak errors in `wget-1.10.2` and 1 leak error in `which-1.16`. Figure 11 shows a memory leak found by Melton. At line `M` in this figure, `read_whole_line` allocates a heap object to `line` which is not freed at line `L`.

```
 fileinfo *ftp_parse_winnt_ls(...)
    ...
M: while (line=read_whole_line(fp)){
     len = clean_line(line);
L:   if (len < 40) continue;
     ... } }
```

Figure 11. Memory leak in file ftp-ls.c of wget-1.10.2.

Two of these bugs in `wget-1.10.2` which also exist in `wget-1.12` are confirmed by the developers of the programs. We analyzed `wget-1.10.2` and `which-1.16` but not the latest release sources so as to compare our work with [1] which only found one bug in `wget` and one bug in `which`.

B. Comparison with Other Tools

To compare with other tools, we have applied Melton to some programs in SPEC2000 that were also analyzed by LC [2], FastCheck [3], SPARROW [4] and WJ2009 [9]. Table I shows the experimental results. The first column shows the names of the programs and the second and third columns describe the number of lines of each program and the analysis time. The following two columns show the real bugs and false positives our analysis tool reported. The last column represents the number of types of false positives. False positives caused by the same code pattern in one program are summarized into one type.

Prog.	Size(kloc)	Time	Bugs	FP	Types of FP
art	1.2	1.7s	0	0	0
bzip2	4.6	7.0s	0	0	0
equake	1.5	0.2s	0	0	0
ammp	13.2	26.3s	14	0	0
gzip	7.7	13.1s	0	0	0
mcf	1.9	10.7s	0	0	0
vortex	52.7	33m29.7s	5	7	2
crafty	18.9	1m31.8s	0	0	0
mesa	49.7	7m38.1s	12	15	3
parser	10.9	2m4.3s	0	0	0

Table I
Experimental results on some programs of SPEC2000.

Fourteen real bugs in `ammp` were reported while LC, FastCheck and SPARROW reported 20 real bugs. It

seems that Melton detects fewer bugs. But after analyzing FastCheck's bug reports, we found that Melton reported one bug with two allocation sites (`a = malloc(32)` and `b = malloc(32)`) in the following code whereas FastCheck reported two bugs. The actual bugs Melton found in `ammp` are the same as other tools.

```
a = malloc(32), b = malloc(32);
c = malloc(32); if (c == NULL) return;
```

Melton detected 5 bugs in `vortex` and 12 in `mesa` that are more than other tools. It benefits from the precise function summaries. Although it generated 7 false positives in `vortex` and 15 false positives in `mesa`, the types of them are 2 and 3 which can be checked easily. Some of them occur due to the weakness of our memory abstraction. That is, Melton cannot capture the side-effect assigning a constant value or symbolic value (not memory object) to pointer arguments. For instance,

```
int *mymalloc(int *state) {
  if (*state == 0) {
    *state = 1; return malloc(64); }
  return NULL; }
void foo() {
  int st = 0; int *m = mymalloc(&st);
  if (st == 1) free(m); }
```

If the function `mymalloc` returns a heap object, the value of `st` should be 1. Melton reports a false positive because it cannot capture the assignment to `*state`. It causes most of the false positives in `vortex`. The poor handling at function pointers also leads Melton to report most of the false positives in `mesa`.

Melton misses some bugs in `art`, `bzip2`, `equake` and `gzip` that other tools can detect. Most of the false negatives are caused by the limited iteration number in loop. We don't check the memory leak that a heap object pointed to by global variables is not freed at the exit of `main` procedure while [9] does and it found the most bugs in the former three programs.

VIII. Related Work

Many tools have been developed for detecting memory leak. Dynamic tools like Purify [10] and Valgrind [11] are commonly used. The programs are instrumented and dynamically executed to trigger the existing memory leak errors. The ability of memory leak detection of dynamic tools depends on the quality of test inputs.

We focus on static detection tools in this paper. The existing memory leak detection tools [1]–[6] using static techniques can be classified into two categories: flow-sensitive and path-sensitive. Generally, flow-sensitive tools have a lower cost of time, but the false positive rate is high. The advantage of path-sensitive tools is the accuracy of bug finding. Most of the existing tools support interprocedural analysis, and are context-sensitive.

To compare with other tools more clearly, we define eleven memory actions classified into four categories that may be captured in analyzing function behaviors in Table II. The first and second column respectively show heap objects' and external objects' escaped actions through arguments, global variables and return value. The following two columns are used to capture the alias in arguments and global variables such as the functions like `strcpy` , `memcpy`, etc that return the argument's alias.

Escaped with heap objects	Freed with external objects	Args' Interprocedural alias	Globals' Inter procedural alias
AllocToArg	ArgToFree	ArgToArg	GlobalToArg
AllocToGlobal	GlobalToFree	ArgToGlobal	GlobalToGlobal
AllocToReturn		ArgToReturn	GlobalToReturn

Table II
11 KINDS OF ACTIONS AND THEIR CATEGORIES

Orlovich and Rugina [2] presented a static memory leak detector called LC that reasons about the absence of bugs by disproving their presence. The algorithm performs a backward dataflow analysis. Clouseau [6] is a flow-sensitive and context-sensitive memory leak detector. It is based on a practical ownership model for managing memory. These two tools generate high false positive rate and are hard to compare with our tool about memory actions.

SPARROW [4] models each procedure into a parameterized summary that is used in analyzing its call sites. Although it makes the summary relevant with return value, path-insensitive analysis causes it to miss some bugs. Besides, it misses some leak bugs that come from interprocedural overwriting of allocated addresses stored in the same global variable because of the weakness in handling memory actions of global variables.

Xie and Aiken [12] developed a context- and path-sensitive static analysis framework, Saturn. The memory leak detector [5] based on Saturn exploits path-sensitivity from modeling the input program as Boolean formulas. The detector is context-sensitive by using summary-based analysis. A summary of a function includes a Boolean value that describes whether the function returns newly allocated memory and a set of escaped locations(escapees). But it cannot capture the memory actions such as *AllocToArg*, *ArgToArg*, etc. That means a heap object escaped by function parameters will be ignored during interprocedural analysis. Melton implements a more precise function summary model than Saturn and is able to detect more kinds of bugs.

FastCheck [3] detects memory leak using a sparse representation of the program in the form of a value-flow graph. The analysis reasons about program behavior on all paths by computing guards for the relevant value-flow edges. But its analysis is not precise enough, for example the actions *AllocToArg*, *AllocArg* are not captured like Saturn. The imprecise and unsound test conditions in guards cause it

to miss some errors.

Xu and Zhang [1] presented a path- and context- sensitive method to detect memory leaks in C programs. Escape analysis is used to summarize the function behavior into 8 categories. The summary representation is concise, but not precise enough. The memory actions about interprocedural alias like *ArgToArg*, *ArgToReturn*, etc are not captured in summaries. Thus it detects fewer bugs than ours. Besides, the summaries do not contain the path conditions for reducing the false positive rate.

Table III shows the difference of the tools in path-sensitivity. The interprocedural path-sensitivity means the path conditions are retained in function summaries and are used in function's call sites.

Tools	Intraprocedural path-sensitivity	Interprocedural path-sensitivity
LC [2]	No	No
Clouseau [6]	No	No
FastCheck [3]	Yes	No
Saturn [5]	Yes	Yes
SPARROW [4]	No	No
XZ08 [1]	Yes	No
Melton	Yes	Yes

Table III
THE DIFFERENCE ABOUT PATH-SENSITIVITY

Table IV describes the memory actions the tools record. It shows that Melton records more memory actions than others. FastCheck captures fewest memory actions, so it detects fewer leaks than Saturn, SPARROW and Melton. But we cannot handle the memory action *ToAlloc* that causes Melton to miss some leaks that other tools can detect.

	Saturn	SPARROW	FastCheck	XZ08	Melton
AllocToArg	No	Yes	No	Yes	Yes
AllocToGlobal	No	No	No	Yes	Yes
AllocToReturn	Yes	Yes	Yes	Yes	Yes
ArgToFree	Yes	Yes	Yes	Yes	Yes
GlobalToFree	Yes	No	No	Yes	Yes
ArgToArg	No	Yes	No	No	Yes
ArgToGlobal	Yes	Yes	No	No	Yes
ArgToReturn	No	Yes	Yes	Yes	Yes
GlobalToArg	Yes	Yes	No	No	Yes
GlobalToGlobal	No	No	No	No	Yes
GlobalToReturn	Yes	Yes	No	No	Yes

Table IV
MEMORY ACTIONS CAPTURED BY MEMORY LEAK DETECTION TOOLS.

IX. CONCLUSION AND FUTURE WORKS

We have presented a novel analysis model called Memory State Transition Graph (MSTG) for detecting memory leak. The model captures procedure behaviors more precisely and reasons about path conditions for reducing false positive rate and path explosion. To support interprocedural analysis, it generates precise summaries which are applied at function's call sites. Several memory leak bugs have been found in real programs and more leak bugs have been detected in some programs of SPEC2000 benchmarks. In some cases Melton produces many false positives, but most of them come from the same code patterns which are easy to check.

Currently, Melton misses some bugs caused by poor handling in loops, recursions and recursive data structures. In the future work, we plan to do optimization in loops such as the loops with constant iteration number and to capture the memory actions in recursive data structures. Besides, we will improve our memory abstraction to describe the function's side-effects more precisely to reduce the false positive rate.

ACKNOWLEDGEMENTS

This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. 61070039 and 61003026. The authors would like to thank an anonymous reviewer for detailed comments.

REFERENCES

[1] Z. Xu and J. Zhang, "Path and context sensitive inter-procedural memory leak detection," in *Proc. of QSIC*, 2008, pp. 412–420.

[2] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *Proc. of SAS*, 2006, pp. 405–424.

[3] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proc. of PLDI*, 2007, pp. 480–491.

[4] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proc. of ISMM*. ACM, 2008, pp. 131–140.

[5] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proc. of ECSE/FSE*, 2005, pp. 115–125.

[6] D. Heine and M. Lam, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," in *Proc. of PLDI*, 2003, pp. 168–181.

[7] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural aliasing," in *Proc. of PLDI*, 1992, pp. 235–248.

[8] "Clang: a C language family frontend for LLVM." http://clang.llvm.org.

[9] J. Wang, X. Ma, W. Dong, H. Xu, and W. Liu, "Demand-driven memory leak detection based on flow and context-sensitive pointer analysis," *JCST*, pp. 347–356, 2009.

[10] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the Winter USENIX Conference*, 1992, pp. 125–138.

[11] "Valgrind," http://www.valgrind.org/.

[12] Y. Xie and A. Aiken, "Scalable error detection using Boolean satisfiability," in *Proc. of POPL*, 2005, pp. 351–363.