Front. Comput. Sci., 2015, 9(1): 34–54
DOI 10.1007/s11704-014-3460-8

RESEARCH ARTICLE

Melton: a practical and precise memory leak detection tool for C programs

Zhenbo XU[1], **Jian ZHANG** (✉)[2], **Zhongxing XU**[2]

1 Department of Computer Science and Technology, University of Science and Technology of China, Anhui 230027, China
2 State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

Abstract Memory leaks are a common type of defect that is hard to detect manually. Existing memory leak detection tools suffer from lack of precise interprocedural analysis and path-sensitivity. To address this problem, we present a static interprocedural analysis algorithm, that performs fully path-sensitive analysis and captures precise function behaviors, to detect memory leak in C programs. The proposed algorithm uses path-sensitive symbolic execution to track 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. In order to do interprocedural analysis, the proposed algorithm generates a summary for each procedure from MSTG and applies the summary at the procedure's call sites. A prototype tool called **Melton** is implemented for this procedure. Melton was applied to five open source C programs and 41 leaks were found. More than 90% of these leaks were subsequently confirmed and fixed by their maintainers. For comparison with other tools, Melton was also applied to some programs in standard performance evaluation corporation (SPEC) CPU 2000 benchmark suite and detected more leaks than the state of the art approaches.

Keywords memory leak, bug finding, static analysis, symbolic execution

Received November 18, 2013; accepted June 24, 2014

E-mail: zj@ios.ac.cn

1 Introduction

Memory leaks are a common type of defect in programs written in languages with explicit memory management, such as C or C++. Memory leaks occur when dynamically allocated memory becomes unreachable but is never freed; this consumes the available memory of the system and reduces system performance. Eventually, the available memory may be exhausted and in the worst case, part or even the whole system stops working correctly.

In recent years there has been a lot of research devoted to detecting memory leaks. The main approaches are dynamic and static analyses. Generally, dynamic analysis is performed by executing instrumented programs to trigger existing bugs. In order to be effective, a target program must be executed with sufficient test inputs with the help of software testing techniques such as code coverage. However, the extra overhead introduced by instrumented instructions slows down the execution of programs and prevents it from being universally applicable. In addition, the detection ability is limited by the quality of test inputs. Different from dynamic analysis, static analysis statically scans the program code and performs a thorough analysis against potential defects existing in programs before run-time. A precise and fine-grained static analysis provides a means to ensure code quality, safety, and dependability, and is especially important for safety-critical software systems.

For static leak detection, a number of tools have been developed [1–9]. They usually track newly allocated heap ob-

jects along with data-flow information to see if the memory storing these objects is correctly released. Some tools like Clouseau [3], LC [5], FastCheck [6], SPARROW [8] and Saber [10] use simpler approaches trading off precision for scalability, including path-insensitivity, limited symbolic computation, or imprecise interprocedural analysis. These trade-offs may not only lead them to miss many bugs, but also introduce numerous false alarms. Other tools applying symbolic analysis to detect memory leaks, including [2, 4, 7], exhaustively traverse all possible execution paths and use an external constraint solver or theorem prover to prune infeasible paths. But these tools suffer from lack of interprocedural alias analysis. They cannot catch the exact points-to relation during procedure invocations, leading to the production of false alarms and false negatives.

In this paper we presents a static interprocedural analysis algorithm using memory state transition graphs (MSTGs) to detect memory leaks in C programs. A memory state transition graph defines a series of states a memory object can have and tracks the state changes during data-flow analysis to check if the leak states can be reached. By using path-sensitive symbolic execution, each function is abstracted to an MSTG. During the abstraction process, memory leaks will be reported if leak states occur. In order to perform interprocedural analysis, function summaries are generated from MSTGs and applied at function call sites. In what follows we use the term *leak* to refer to a memory leak, unless otherwise specified.

In contrast to the simpler approaches, to gain a more precise analysis, our algorithm is path-sensitive; this can effectively eliminate false positive reports and provide more helpful information about what path segments lead to memory leaks. To address the scalability of path-sensitive analysis for large programs, summary-based analysis is performed to avoid re-analysis of the functions that are called.

Compared with approaches that use symbolic analysis, our method provides a fine-grained abstraction to capture more precise function behaviors. We use a region based memory model [11] to statically simulate the memory state at each program point when performing intraprocedural analysis. We extract precise memory actions from this model for handling global pointer aliases and interprocedural analysis; some of these memory actions are ignored by other approaches.

The main features of our approach include:

- Concise memory abstraction is used to model memory objects related to memory leaks as *heap objects* and *external objects*. Heap objects denote memory allocated in the heap space and are checked for the existence of leaked state. External objects represent the memory space that is unknown inside a procedure (such as a memory object passed in by a global pointer) and provide alias information during interprocedural analysis. Generally, *memory objects* in this paper refer to heap objects or external objects.
- A practical analysis model called MSTG is proposed to analyze C programs. Each MSTG is generated from a function using path-sensitive symbolic execution. The states of memory objects contained in MSTG nodes indicate whether leaked states exist in the procedure. Memory actions that cause state transitions can precisely capture the function behaviors. Predicates, namely path conditions annotated on MSTG's edges, are checked for satisfiability to reduce the number of false alarms.
- Precise procedure summaries generated from MSTGs consist of memory actions and path conditions. When applying the summaries, memory actions are used to capture the procedural side-effects and path conditions are checked for satisfiability to prune the impossible memory actions.
- A helpful bug reporter is used to provide rich diagnostic information and guidance for correcting memory leaks. A main challenge in applying static tools is the high manual cost of confirming the leak reports, especially when the false positive rate is high. In addition, memory leaks have few visible symptoms other than the slow and steady increase in memory consumption. Our bug reporter identifies not only allocation sites and leaking statements, but also the function call trace of allocation sites and path segments that lead to the memory leaks.

In order to investigate the capabilities of our approach for detecting memory leaks, we implemented a prototype tool called **Melton** and applied it to five programs from two libraries `sqlite` and `libosip` that are often called by long running systems, a SSH connectivity tool `OpenSSH` and two packages from GNU. Melton discovered 41 leaks in these programs, more that 90% were new, and were subsequently all confirmed and fixed by the maintainers of the programs. We also use it to detect memory leaks in some programs from standard performance evaluation corporation (SPEC) CPU 2000 benchmarks. Our experimental results show that some extra leaks were found by our tool while other tools missed them [5, 6, 8, 10].

However, we would like to emphasize that, as a static analysis tool, our tool still has several limitations. For example, loops may introduce a large number of program paths that may lead the analysis to be very time-consuming. To address this problem, we unroll each loop body for just a few times (two iterations in our implementation). Recursions and function pointers are processed by performing a conservative analysis that invalidates variables that may be changed. For interprocedural aliasing, difficulties still remain in applying a sound method in the sense of precision, extensibility, and scalability. As our goal is bug-finding but not sound verification, we assume parameters in one function do not have any alias. Currently, our tool does not consider the effects of interleaving in multi-threaded programs.

In summary, the contributions of this paper include:

- A practical and precise abstraction and analysis model for performing intraprocedural analysis.
- Procedure summary generation and application with precise memory actions and path conditions for capturing interprocedural side-effects.
- Encouraging experimental results with 41 detected leaks in five commonly used open source projects and hundreds of leak discovered in SPEC CPU 2000.

Related work is discussed in Section 2. Section 3 gives several motivating examples. In Section 4 we describe our main approach, more specifically: Section 4.1 describes our algorithm framework; Section 4.2 introduces our intraprocedural analysis model including memory abstraction and memory state transition graph; Section 4.3 describes the interprocedural analysis including summary generation and summary application; and an illustrative example is given in Section 4.4. The detailed implementation is given in Section 5. Experimental results are shown in Section 6. Finally, we give our conclusion and future work in Section 7.

2 Related work

Many tools have been developed for detecting memory leaks. Dynamic analysis tools like Purify [12] have been commonly used. Such programs are instrumented and dynamically executed to trigger memory leaks. Although it is effective in detecting memory leaks, the extra overhead introduced by instrumented instructions slows down the execution of the programs and prevents it from being universally applicable, e.g., in embedded systems.

Recently, a large body of work [13–18] has been proposed for detecting Java memory leaks. Dynamic analysis is used by introducing growing types [13,16] or object staleness [14,15] to identify suspicious data structures that may contribute to a memory leak. The same dynamic analysis limitations also apply to these approaches.

The static approach [18] proposes a separation logic and bi-abductive inference based algorithm for identifying memory leak in Java. All of the approaches mentioned above detect memory leaks that occur when references to unused objects are unnecessarily held so that these objects cannot be garbage collected whereas we aim to detect leaks caused by objects not reachable from program variables.

Some other work like [19–21] uses concolic execution, i.e., combining symbolic execution and concrete execution, to automatically generate test cases, attempting to achieve high coverage of program executions or trigger system bugs. These tools are effective in detecting system defects that cause the system to crash immediately. However, since memory leaks have few visible symptoms other than the slow and steady increase in memory consumption, even a test case triggers memory leaks, the leaks are hard to find.

In this paper, we focus on static detection tools for languages with explicit memory management. Existing memory leak detection tools of this kind [3–8, 10] can be classified into two categories: path-insensitive and path-sensitive. Generally, path-insensitive tools have a lower time cost, but the false positive rate is high; this is one of the main challenges for static analysis tools. The advantage of path-sensitive tools is the accuracy of leak finding. Symbolic analysis is usually performed for each path in programs to check the satisfiability of the path condition to reduce false alarms. More specifically, path-sensitivity can be either both intraprocedural or interprocedural. For summary-based analysis, interprocedural path-sensitivity means function effects guarded with a path condition on each path are retained separately in function summaries and are applied at function call sites. Table 1 shows the difference between these tools in path-sensitivity. Most of these tools including [3, 5, 6, 8, 10] perform totally path-insensitive analysis. XZ08 [7] applies intraprocedural path-sensitivity but is not interprocedural path-sensitive. Only Saturn [4] and our tool **Melton** perform fully path-sensitive analysis and gain better precision than other tools.

In addition to path-sensitivity, in order to obtain higher precision, making fine-grained memory actions is also instrumental in capturing function behaviors. To compare with other tools more clearly, we define 14 kinds of memory actions classified into four categories in Table 2. The first col-

umn shows the category names, the second column gives the action names and the last column represents whether false positives or false negatives will be caused if we do not capture the actions interprocedurally, where **FP** and **FN** represent false positive and false negative, respectively.

Table 1 The difference on path-sensitivity

Tool	Intraprocedural path-sensitivity	Interprocedural path-sensitivity
LC [5]	No	No
Clouseau [3]	No	No
Saturn [4]	Yes	Yes
FastCheck [6]	No	No
SPARROW [8]	No	No
XZ08 [7]	Yes	No
Saber [10]	No	No
Melton	Yes	Yes

Table 2 14 kinds of actions classified in different categories

Category Name	Action Name	FN/FP
Escaping heap objects	AllocToArg	FN
	AllocToGlobal	FN
	AllocToReturn	FN
	AllocToAlloc	FN
Interprocedural alias in Args	ArgToArg	FP
	ArgToGlobal	FP
	ArgToReturn	FP
	ArgToAlloc	FP
Interprocedural alias in Globals	GlobalToArg	FP
	GlobalToGlobal	FP
	GlobalToReturn	FP
	GlobalToAlloc	FP
Freeing external objects	ArgToFree	FP
	GlobalToFree	FP

Escaping heap objects category represents heap objects escaping via arguments, global variables, return value and other escaped heap objects. Figure 1(a) gives four examples describing each action. The former three functions (or actions) are commonly used methods to pass out newly allocated heap objects. The final function (action) *AllocToAlloc* represents the second allocated heap object escaped by the `next` field of the first allocated heap object. This action usually involves recursive data structures such as lists, since most recursive data structures are organized by fields of one heap object pointing to another heap object. The following two categories are used to capture aliases in arguments and global variables such as functions like `strcpy` and `memcpy` that return the argument's alias (e.g., *ArgToReturn*). The relevant examples are given in Figs. 1(b) and 1(c). The forth category of Table 2 gives free actions in external objects (i.e., memory objects passed in by pointers of function parameters and global variables), corresponding to Fig. 1(d).

```
void AllocToArg (void **arg){
   *arg = malloc (32);
}
void *g;
void AllocToGlobal (){
   g = malloc (32);
}
void *AllocToReturn (){
   return malloc (32);
}
list* AllocToAlloc () {
   list *l=
     (list *) malloc (…);
   l->next=
     (list *) malloc (…);
   return l;
}
```

(a)

```
void ArgToArg(void *a,
                void **b){
   *b = a;
}
void *g;
void ArgToGlobal (void *a){
   g = a;
}
void *ArgToReturn (void *a){
   return a;
}
list* ArgToAlloc (list *a){
   list *l=
     (list *) malloc (…);
   l->next = a;
   return l;
}
```

(b)

```
void *g1, *g2;
void GlobalToArg (void *a){
   g1 = a;
}
void GlobalToGlobal (){
   g1 = g2;
}
void *GlobalToReturn (){
   return g1;
}
void GlobalToAlloc (){
   list *l = (list *) malloc (…);
   l->next = (list *) g1;
   …
}
```

(a)

```
void ArgToFree (void *p){
   if (p! = NULL) free (p);
}
void *g;
void GlobalToFree (){
   if (g! = NULL) free (g);
}
```

(b)

Fig. 1 Simple examples of memory actions. (a) Escaping heap objects; (b) interprocedural alias in Args; (c) interprocedural alias in Globals; (d) freeing external objects

These actions describe most of the behaviors a function can have. Imprecision, such as false positive or false negative, may be caused by missing some of these actions. The more actions that can be captured, the more precise the analysis can become[1)]. As shown in the third column in Table 2, missing actions in the first category can lead to false negatives because allocated heap objects are omitted and leaks rooted from these heap objects cannot be detected. For example, if *AllocToArg* is not captured, a leak in the following code will be missed:

```
char *p; AllocToArg(&p); p = NULL,
```

For other actions, missing any of them may cause false alarms. Take *ArgToReturn* as an example, given the following code snippet:

```
p=malloc(...);r=ArgToReturn(p);free(r),
```

[1)] An action being *captured* indicates that this action is recorded during intraprocedural analysis and can be reused during interprocedural analysis.

a false positive will be raised with message like "heap object allocated to `p` is not freed" if the alias between `p` and `r` is not captured. Another explicit example is the missing of free actions (the fourth category), as a freed heap object will be treated as an alive one.

In the following, we discuss other tools for these actions in detail and give a summary table about what actions these tools can capture.

Orlovich and Rugina [5] presented a static memory leak detector called LC that reasons about the absence of memory leaks by disproving their presence. The algorithm performs a backward data-flow analysis. Clouseau [3] is a flow- and context-sensitive memory leak detector. It is based on a practical ownership model for managing memory. These two tools generate a high false positive rate and are hard to compare with our tool on memory actions since their approaches are not concerned with memory actions.

Xie and Aiken [22] developed Saturn, a context- and path-sensitive static analysis framework. A memory leak detector based on Saturn [4] 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* and *ArgToArg*. 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 memory leak.

FastCheck [6] detects memory leaks using a sparse representation of the program in the form of a value-flow graph. The analysis reasons about program behaviors on all paths by computing guards for the relevant value-flow edges. But its analysis is too imprecise, for example the actions *AllocToArg*, *ArgToArg* are not captured. Saber [10] is an improved version of FastCheck. It builds a more precise sparse representation to capture *ToArg* and *ToAlloc* actions but it still can not handle actions related with global variables.

SPARROW [8] models each procedure as a parameterized summary that is used in analyzing its call sites. Although it makes the summaries relevant with return values, path-insensitive analysis leads it to miss some leaks. Besides, it misses some leaks that come from interprocedural overwriting of allocated addresses stored in the same global variable because of the weakness of handling memory actions of global variables. It means the actions of global variables such as *ArgToGlobal* and *GlobalToGlobal* can not be captured.

Xu and Zhang [7] presented a path- and context-sensitive method to detect memory leaks in C programs. Escape analysis is used to summarize the function behaviors into eight categories. The summary representation is concise, but not precise enough. The memory actions about interprocedural alias like *ArgToArg* and *ArgToReturn* are not captured in summaries. Thus it detects fewer leaks than our approach.

The memory actions captured by the above mentioned tools are summarized in Table 3, where **SW.** is short for **SPARROW** (We do not list **FastCheck** because **Saber** is an improvement on it.). As shown in this table, **Melton** records more memory actions than the others, except for actions about *ToAlloc*, i.e., *AllocToAlloc*, *ArgToAlloc*, and *GlobalToAlloc*. Currently, we cannot apply summaries that contain *ToAlloc* to caller context because our model cannot track the states of heap objects that are assigned to other heap objects. This leads to Melton missing some leaks that SPARROW and Saber can detect. Nonetheless, Melton performs a fully path-sensitive analysis to gain more precise detection than path-insensitive tools, and a more fine-grained memory abstraction than other path-sensitive tools.

Table 3 Memory actions captured by memory leak detection tools

	Saturn	SW.	XZ08	Saber	Melton
AllocToArg	No	Yes	Yes	Yes	Yes
AllocToGlobal	No	No	Yes	No	Yes
AllocToReturn	Yes	Yes	Yes	Yes	Yes
AllocToAlloc	No	Yes	No	Yes	No
ArgToArg	No	Yes	No	Yes	Yes
ArgToGlobal	No	Yes	No	No	Yes
ArgToReturn	No	Yes	No	Yes	Yes
ArgToAlloc	No	Yes	No	Yes	No
GlobalToArg	Yes	Yes	No	No	Yes
GlobalToGlobal	No	No	No	No	Yes
GlobalToReturn	Yes	Yes	No	No	Yes
GlobalToAlloc	No	No	No	No	No
ArgToFree	Yes	Yes	Yes	Yes	Yes
GlobalToFree	Yes	No	Yes	No	Yes

3 Motivation and examples

This section will use several examples to illustrate the advantages of being path-sensitive in Section 3.1 and the necessity of recording memory actions in Section 3.2. Finally, a comprehensive example will be given in Section 3.3 to show the weaknesses of existing tools.

3.1 Path-sensitivity

A major weakness of static analysis tools is that they produce a high false positive rate. Being path-sensitive can ef-

fectively reduce false alarms by pruning infeasible paths. We use two examples to show the effectiveness of intra- and inter-procedural path-sensitive analysis in Sections 3.1.1 and 3.1.2, respectively.

3.1.1 Intraprocedural path-sensitivity

Figure 2 shows a leak-free function wherein a path-insensitive analysis will produce false positive. In this example, a heap object is allocated to `buf` at line 4 if the argument *len* is greater than `0`. This heap object is freed with the same condition at line 7. Figure 3 gives the control flow graph of function `foo`. During state merging in path-insensitive analysis, the value of `buf` may refer to a heap object allocated by `malloc`(*len*) or `NULL` at block *b*4. This will produce a false alarm since the analysis treats the heap object pointed to by `buf` as not freed over the path $b1 \rightarrow b2 \rightarrow b4 \rightarrow b7 \rightarrow b8$.

```
1  void foo (int len) {
2    char *buf;
3    if (len > 0)
4      buf = (char *) malloc (len);
5    else buf = NULL;
6    …
7    if (len > 0) free (buf);
8    else …
9    …
10 }
```

Fig. 2 A simple example for illustrating intraprocedural path-sensitive analysis

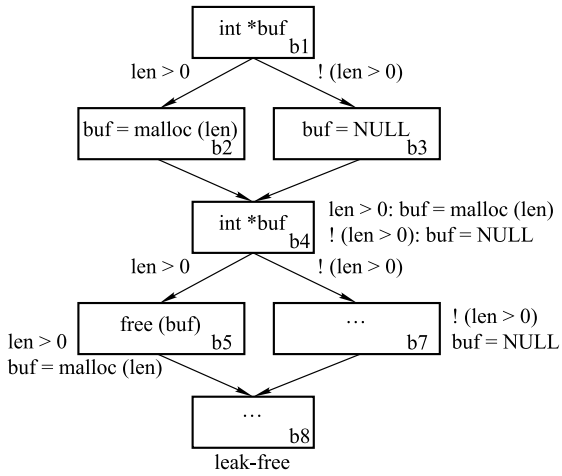

Fig. 3 The CFG of function foo

For path-sensitive analysis, the value of `buf` is guarded by path constraints in different execution paths. As shown in block *b*4 of Figure 3, if $len > 0$, `buf` points to `malloc`(*len*). Otherwise, the value of `buf` is `NULL`. The former value will be eliminated when the analysis reaches block *b*7 since the constraint in $b4 \rightarrow b7$ is $!(len > 0)$. So it is leak-free over the path segment $b4 \rightarrow b7 \rightarrow b8$. Similarly, it is safe over path segment $b4 \rightarrow b5 \rightarrow b8$. In fact, the path $b1 \rightarrow b2 \rightarrow b4 \rightarrow b7 \rightarrow b8$ that leads to the false alarm in path-insensitive analysis is an infeasible path that is pruned by checking the satisfiability of the path condition in path-sensitive analysis.

3.1.2 Interprocedural path-sensitivity

Most existing memory leak detection approaches rely on summary-based analysis. In their summaries, all possible paths are combined into a single state called meet-over-all-paths (MOP) [23]. MOP may also introduce imprecision in interprocedural memory leak detection. Consider Fig. 4, function `alloc_or_global` returns a heap object when *len* is greater than `MAX`, otherwise it returns the global buffer. Function `foo` calls it and frees the returned object when *len* is greater than `MAX`. `foo` is a leak-free function. However, the approaches using MOP merge the two paths in `alloc_or_global` into one state, i.e., the return value is `malloc`(*len*) or `global_buf`. A false alarm will be produced in function `foo` as it considers the `else` branch at line 12 does not free the allocated buffer.

```
1  char global_buf [MAX];
2  char *alloc_or_global (unsigned len) {
3    if (len > MAX)
4      return (char *) malloc (len);
5    else
6      return global_buf;
7  }
8  void foo (unsigned len) {
9    char *buf = alloc_or_global (len);
10   … //use buf
11   if (len > MAX) free (buf);
12   else …
13 }
```

Fig. 4 An example for illustrating interprocedural path-sensitive analysis

In our approach, when performing interprocedural path-sensitive analysis, the constraints in summaries are retained and the analyzer precisely determines that the pointer `buf` points to `global_buf` in the `else` branch on line 12 and there is no need to free `buf`.

3.2 Memory actions

As described in Section 2, memory actions are instrumental in capturing function behaviors for finding more potential bugs and eliminating false alarms. Two examples are given here to illustrate the importance of memory actions.

3.2.1 Memory allocating and freeing actions

Memory allocating (including *AllocToArg*, *AllocToGlobal*, *AllocToReturn* and *AllocToAlloc*) and freeing actions (including *ArgToFree* and *GlobalToFree*) are basic memory management operations that are accomplished using library functions, such as `malloc`, `realloc` and `free` in the C Programming language. But more often, programmers prefer to write wrapper functions to achieve some other operations such as initialization or manipulating allocation result (success or failure).

Figure 5 gives a leak example found by our tool in `OpenSSH-5.9p1`. Function `buffer_init` initializes `buffer->buf` with a newly allocated object that can be freed by function `buffer_free` passed by argument. A leak occurs if function `mux_client_read` returns nonzero. In this example, function `buffer_init` involves *AllocToArg* action and function `buffer_free` involves *ArgToFree* action. Approaches without capturing *AllocToArg* action such as Saturn [4] and FastCheck [6] will miss this kind of leak.

```
void buffer_init (Buffer *buffer){
   …
   buffer->buf = xmalloc (len);
}
void buffer_free (Buffer *buffer){
   …
   xfree (buffer->buf);
}
int mux_client_read_packet (…){
   Buffer queue;
   buffer_init (&queue);
   …
   if (mux_client_read (…) != 0){
     return −1; // leak point
   }
   buffer_free (&queue);
   return 0;
}
```

Fig. 5 Leak found in OpenSSH-5.9p1 related with allocating and freeing actions

3.2.2 Interprocedural aliasing

Interprocedural aliasing attempts to determine if two (pointer) expressions refer to the same storage location. Although many techniques have been proposed for computing interprocedural aliasing, difficulties still remain in applying such methods in the sense of precision, extensibility, and scalability.

Since we focus on static memory leak bug-finding tools but not verification tools that should be sound, rather than computing a sound but impractical alias set, we extract eight actions (the last two categories in Table 2) for manipulating interprocedural aliasing. They are not sound or complete but are effective in eliminating false positives in static memory leak detection. For example, Figure 6 is a false positive from Saturn [4]. At line 3, `BN_copy` returns the alias of `t` on success (related to *ArgToReturn* action) and `NULL` on failure. At line 4, when `r` is `NULL` indicating that `BN_copy` failed, the allocated heap object that `t` points to is released. Otherwise, the alias of `t`, i.e., `r` is returned. Therefore, no leak exists in this example. Approaches that cannot distinguish between `t` and its alias `r` after `BN_copy` is called may raise false alarms.

```
1 t = BN_new ();
2 if (t == NULL) return NULL;
3 r = BN_copy (t, a);
4 if (r == NULL) BN_free (t);
5 return r;
```

Fig. 6 A false positive from [4]

3.3 A comprehensive example

The ability of existing tools is greatly limited by path-insensitivity and imprecise summaries of function behaviors. The code fragment in Fig. 7, which is used to describe the necessity of recording precise memory actions and path conditions, is a simplified version from some real 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 accessed and modified through the second parameter `char **p`. Functions `foo1` and `foo2` call `bufSelect` to get a buffer.

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

Fig. 7 A comprehensive example with memory leak

Consider function `foo1`, at line 9 `bufSelect` is called. Since the value of variable *len* is greater than `10`

(`bufSelect` line 3), a heap object is allocated to `buf` at line 4. A memory leak occurs at the exit of `foo1` when the heap object is not released. Saturn [4] and FastCheck [6] miss this leak because it cannot handle function behaviors wherein heap objects are escaped by parameters.

Function `foo2` does not have a memory leak but some tools [3, 5, 7, 8, 10] detect a false positive leak. The call to `bufSelect` at line 15 just assigns `fastbuf` to `buf` (`bufSelect` line 3) and no heap memory is allocated. Since the function `bufSelect` contains two program paths, one of which allocates a heap object and the other one does not, the tools *without intraprocedural path-sensitivity* [3, 5, 8, 10] merge these two paths into a single allocation path meaning that the call site at line 15 is analyzed as an allocation site. Thus false leak detection will be produced by these tools. The tool [7] *without interprocedural path-sensitivity*, i.e., merging all of memory behaviors in different paths at the exit of functions, also summarizes the function `bufSelect` as an allocation path and produces false leak detections.

4 Leak detection using MSTG

This section first describes the algorithm framework of our leak detection approach in Section 4.1 and then presents intraprocedural and interprocedural analysis in detail in Section 4.2 and Section 4.3, respectively. An illustrative example is given in Section 4.4. Finally, we discuss the advantages and some potential optimizations using memory state transition graph in Section 4.5.

4.1 Algorithm framework

Algorithm 1 gives the framework of our algorithm *LeakDetect*. The 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 ensure each called function has been analyzed. The *Visit* method first generates an MSTG for function f and reports the detected leaks during generation. Finally, a summary Sum_f of function f is generated from the built MSTG and function f is marked as visited. *GenMSTG* and *GenSum* methods in *Visit* are described in subsequent sections.

Since recursion and function pointers cannot be handled, we process calls with recursive functions and function pointers in a conservative way. That is, variables such as function parameters or return values (we view return values as abstract variables) that may be changed by callees are assigned unknown or symbolic values.

Algorithm 1 Leak detection algorithm

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 for**
5: $Sum_f = GenSum(MSTG_f)$

4.2 Intraprocedural analysis

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

4.2.1 Memory abstraction

4.2.1.1 Region classification

We classify memory regions into four sets: *Local*, *Static*, *Heaps*, and *Externs*, during intraprocedural analysis. Memory regions allocated in stack are represented by the set *Locals*. The set *Statics* includes variables stored in static space such as global variables and static variables. All remaining memory excluding those in *Locals* ∪ *Statics* are abstracted into *heap objects* and *external objects*. Heap objects are the memory objects that are dynamically allocated by library functions such as *malloc* and *realloc*. We use the set *Heaps* to represent heap objects. External objects are memory objects passed in by pointers of function parameters and global variables, represented with the set *Externs*.

We provide a clarifying example in Fig. 8 that extracts only the `bufSelect` function from Fig. 7 to demonstrate the above classification. First, we use the notation $Reg(p)$ to represent the memory object that pointer `p` points to and $Base(v)$ to represent the memory object that variable v is stored in. So, in Fig. 8 we have:

$Locals = \{Base(len), Base(p), Base(\text{fastbuf})\}$
$Externs = \{Reg(p), Reg(*p), Reg(fastbuf)\}$
$Heaps = \{malloc(len)\}$
$Static = \{\}$

Note that since memory abstraction is limited in the context of the current function, these four sets are disjoint. For example, given a global variable `char *g`, $Base(g)$ belongs to *Statics* while $Reg(g)$ (which is equal to $Base(*g)$) belongs

to *Externs* (Supposing that the memory region `g` points to is unknown).

```
1  void bufSelect (int len, char **p,
2                  char *fastbuf) {
     //ToExtern:Reg (fastbuf) ->*p
3    if (len <= 10) *p = fastbuf;
     //ToExtern:HeapObj->*p
4    else *p = (char *) malloc (len);
5  }
```

Fig. 8 A simple example illustrating memory abstraction

4.2.1.2 Action abstraction

According to the above memory region classification, we define the following six actions that operate on each heap and external memory object.

- *ToLocal* assigns memory objects to pointers whose base regions belong to *Locals*. For instance, given `void f(char *p){ char *q=p; }`, memory object *Reg*(p) is assigned to q which is a local variable.
- *ToExtern* assigns memory objects to pointers whose base regions belong to *Externs* or *Statics*. This contains the *ToArg* action and *ToGlobal* action. As shown in Figure 8, `*p = fastbuf` can be extracted to the *ToExtern* action of *Reg*($fastbuf$) since *Base*($*p$) belongs to *Externs*.
- *ToReturn* returns a heap or external memory object. For example, given `void* f(void *p) { return p; }`, external memory object *Reg*(p) is returned.
- *ToAlloc* assigns memory objects to pointers whose base regions are heap objects. For instance, given `list *l = (list *)malloc(...)`, programs may assign a newly allocated node to `l->next`, i.e. `l->next=(list*)malloc(...)`. Here, `l->next` is a pointer whose base region is a heap object and the heap object `l->next` points to has *ToAlloc* action.
- *ToFree* frees a memory object.
- *ToUnknown* assigns memory objects to unknown pointers, such as an element of the array of pointers with symbolic index.

Apart from ToLocal and ToUnknown, the actions are consistent with the memory actions listed in Section 2. *ToLocal* action is useless and should be discarded during interprocedural analysis because local variables are destroyed when they pass out of scope of the current function. *ToUnknown* is used to eliminate false positives. It is hard to determine what kind of expressions other tools treat as unknown. So we do not include the *ToUnknown* action in the comparison in Section 2.

4.2.1.3 State abstraction

States of heap objects and external objects are affected by the above actions. We define the following five memory states for each heap object.

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

We also present four memory states for each external object.

- *Accessed* external objects are initialized and are only pointed to by local or temporary pointer variables in the context of the current function.
- *Escaped* external objects are returned or assigned to argument pointers or global pointers (including *ToReturn* and *ToExtern* actions).
- *Freed* external objects are freed (including *ToFree* action).
- *Relinquished* external objects are assigned to some complex C expressions that we cannot handle (including *ToUnknown* action).

4.2.2 Basic state transition model

Based on the memory abstraction above, we define a basic state transition model (BSTM) for a single memory object as $\langle S, L, \rightarrow \rangle$, where

- S is a set of memory states. It consists of states of heap objects and external objects. *Allocated* and *Accessed* are two initial states. The others can be treated as terminal states that represent the exit states of memory object.

- L is a set of labels. A label l in L can be a predicate that represents a set of symbolic conditional expressions extracted from the procedure's path conditions using symbolic execution, or an action that manipulates memory objects.
- $\rightarrow \subset S \times L \times S$ is a ternary relation of labeled transitions. The definition of $\rightarrow$ is a tuple $\langle cs, l, ns \rangle$ where cs and ns respectively denote the current state and the next state of the transition. This can be written as $cs \xrightarrow{l} ns$ which represents the transition from state cs to state ns guarded by label l, i.e., a predicate or a memory action (a memory action can be considered to be a post predicate).

We define BSTM for a single heap object and external object in Figs. 9 and 10 respectively. These two figures describe the transition relations between states.

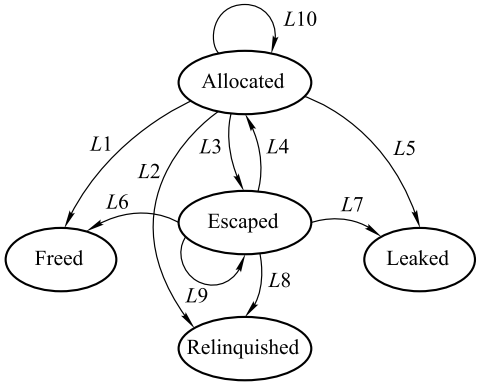

Fig. 9 BSTM for heap objects

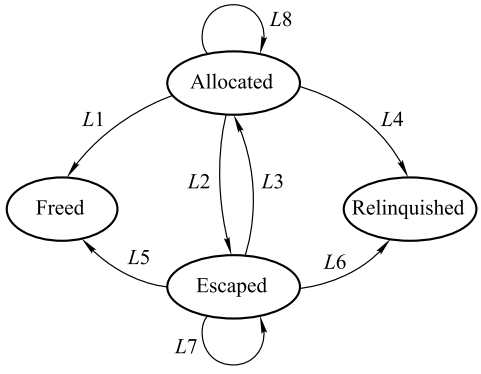

Fig. 10 BSTM for external objects

In Fig. 9, the state of a newly allocated heap object is *Allocated*. It can become itself by *ToLocal* action or *Escaped* state when the label is *ToExtern* or *ToReturn* action. With *ToFree* or *ToUnknown* action, *Allocated* state will become *Freed* or *Relinquished*, respectively. There are two kinds of condition where the *Leaked* state will occur: no reference by which a heap object can be reached, called *lost memory*, and remaining in the *Allocated* state at the exit of function, called *forgotten memory* [24]. *Escaped* state can transition other states just as *Allocated* state does. And it will become a *Leaked* state when *lost memory* occurs.

In Fig. 10, the initialized state of an external object is *Accessed*. Mostly, the transition relation in this figure is similar to Fig. 9. The main difference is that an external object does not contain *Leaked* state.

Yet BSTM is not enough to track the behaviors of a program precisely. Consider label $L4$ in Fig. 9 (or label $L3$ in Fig. 10), for a single memory object, state transition $L4$ relies on the actions of other objects. For example, consider the following code:

```
1 char *g;
2 void foo(char *a) {
3     ...
4     char *p = (char *)malloc(plen);
5     g = p;
6     ...
7     g = a;
8 }
```

in which `p` is a local pointer variable that points to a newly allocated heap object H_p at line 4 and `g` is a global pointer variable that also points to H_p after line 5. At this point, H_p has the *Escaped* state. At line 7, `a` is assigned to `g` producing the *ToExtern* action of `a` and state of H_p will become *Allocated*. Meanwhile, the state of external object E_a introduced by `a` will become *Escaped*. Therefore, more than one BSTM may be affected by a single action. It is hard to describe the relations between different memory objects using BSTM alone.

4.2.3 MSTG

4.2.3.1 Definition

In order to build a more precise model for state transitions in different memory objects, we introduce a more expressive model MSTG based on BSTM; MSTG can be used to represent the abstraction of a program. It is defined as $\langle S, M, L, \rightarrow \rangle$, where

- S is a set of memory states same as that in the definition of BSTM.
- M is a set of memory objects that contains heap objects and external objects.
- L is a set of labels, just as in the definition of BSTM.

- $\rightarrow \subset 2^{M\times S} \times L \times 2^{M\times S}$ is a ternary relation of labeled transitions.

An MSTG is shown in Fig. 11. Each node contains a set of memory objects and their corresponding states i.e., $2^{M\times S}$. Each edge annotated with a label in L denotes a transition between two nodes guarded by this label. A transition may change states of several memory objects in one node. The state changing should comply with the transition rules of BSTM.

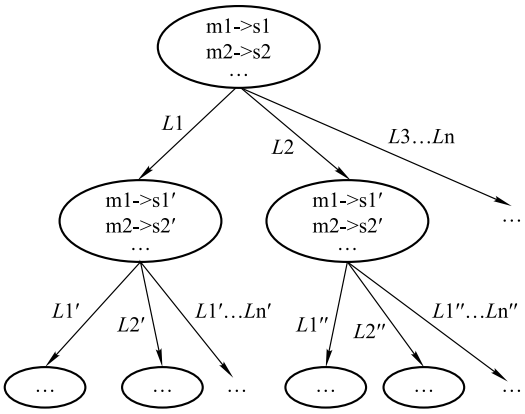

Fig. 11 MSTG

4.2.3.2 MSTG generation

We use path-sensitive symbolic execution to construct memory state transition graphs from a program. Algorithm 2 describes the generation algorithm. We first build a control flow graph (CFG) for function f, and use a breadth-first search (BFS) algorithm to traverse the CFG. We have also implemented a depth-first search (DFS) algorithm in our tool. Since these two algorithms are similar, we only describe the BFS here. An MSTG is generated during the traversal.

When analyzing each block, we are concerned with assignment statements, free statements, return statements, call statements, and branch statements. The three former types of statements may cause state transitions. We first obtain the target memory object from the statement. For example, given the assignment statement `q = p` from pointer `p` to pointer `q`, the target memory is *Reg*(p) and the action is determined by what kinds of region set *Base*(q) belongs to. The *UpdateState* function creates a new node *NewN* from node *N* with the annotated edge, denoted by $N \xrightarrow{Act} NewN$. If a leaked state exists during the update process, leaks will be added to the reporter.

Summaries are applied at call statements. The algorithm *ApplySum* will be given in the next section. Branch statements generate two new child nodes to represent the *true* and *false* branches. The satisfiability of the path condition (PC) is checked and a new node is created with edge $N \xrightarrow{PC} NewN$. We do not list the handling of switch statements in the algorithm for clarity.

Algorithm 2 The algorithm of MSTG generation

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

BFS(Block *B*, Node *N*)
1: **for all** stmt (statement) *s* in *B* **do**
2: do symbolic execution
3: **if** *s* is an assignment stmt from pointer to pointer **then**
4: get target memory object *m* and extract action *Act* from *s*
5: *NewN* = *UpdateState*(*N*, *m*, *Act*), *N* = *NewN*
6: **else if** *s* is a return stmt that returns a memory object *m* **then**
7: *NewN*=*UpdateState*(*N*, *m*, *ToReturn*), *N* = *NewN*
8: **else if** *s* is a free stmt with target memory object *m* **then**
9: *NewN*=*UpdateState*(*N*, *m*, *ToFree*), *N* = *NewN*
10: **else if** *s* is a call stmt and cf is the callee function **then**
11: *ApplySum*(Sum_{cf}, *N*)
12: **else if** *s* is a branch stmt with condition *PC* **then**
13: **if** *PC* is satisfiable **then**
14: *LN*=*NewNode*(*N*, *PC*), BFS(*LB*, *LN*)
15: **end if**
16: **if** !*PC* is satisfiable **then**
17: *RN*=*NewNode*(*N*, !*PC*), BFS(*RB*, *RN*)
18: **end if**
19: **end if**
20: **end for**

4.2.3.3 MSTG generation example

Figure 12 gives a simple example function `foo` for showing MSTG generation. The CFG generated for function `foo` is shown in Fig. 13. We use *Nop* block to split `if-else` statements that contain more than three branches. This program

```
1  void *g;
2  void *foo (int x) {
3    x = x + 1;
4    void *p = malloc (32);
5    if (x < 6)
6      return p;
7    else if (x < 9) {
8      g = p;
9      return NULL;
10   } else {
11     free (p);
12     return NULL;
13   }
14 }
```

Fig. 12 A simple example showing MSTG generation

first allocates a heap object, and has three different paths with different path conditions at the exit of the function. The heap object at each path has its own state: *Escaped* state (by return value) at line 6, *Escaped* state (by global pointer) at line 8 and *Freed* state at line 11. The memory state transition graph is shown in Fig. 14.

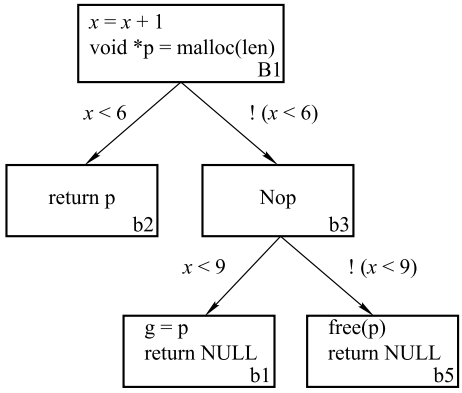

Fig. 13 The CFG of function foo

To generate a memory state transition graph, we first track the control flow graph and use symbolic execution to substitute the expressions. At line 3, the parameter of function `foo` is assigned a unique symbolic identifier $\$x$. After line 3 is "executed", the value of variable x is $\$x + 1$. So we get the symbolic conditional expressions as path condition $PC : (\$x + 1) < 6$ and $PC :!((\$x + 1) < 6)$ annotated at edge $S1 \rightarrow S2$ and edge $S1 \rightarrow S4$ in Fig. 14. In $S1$, the state of $Reg(g)$ is initialized as *Accessed* state and *HeapObj1* denotes the heap object allocated at line 4 in Fig. 12. The action *HeapObj1* $\rightarrow$ *return* annotated at edge $S2 \rightarrow S3$ causes state of *HeapObj1* to become *Escaped*. At $S6$, $Reg(g)$ is removed as no pointer points to it in this function.

The generation of the memory state transition graph is actually an intraprocedural detection. Suppose that the above code does not have the statement `free(p)` at line 11, thus it reaches a leaked state at the third branch when exiting this function. We perform a post check at the end of each path. If there still exist heap objects with *Allocated* state, these objects are treated as leaks.

4.3 Interprocedural analysis

This section describes the summary-based approach to interprocedural leak detection. Section 4.3.1 defines the summary representation and discusses how to generate a summary from an MSTG. Section 4.3.2 shows a summary application at function call sites.

4.3.1 Summary generation from MSTG

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

The following definition describes the summary representation:

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

where P_i denotes the predicate and $UActs_i$ denotes a union

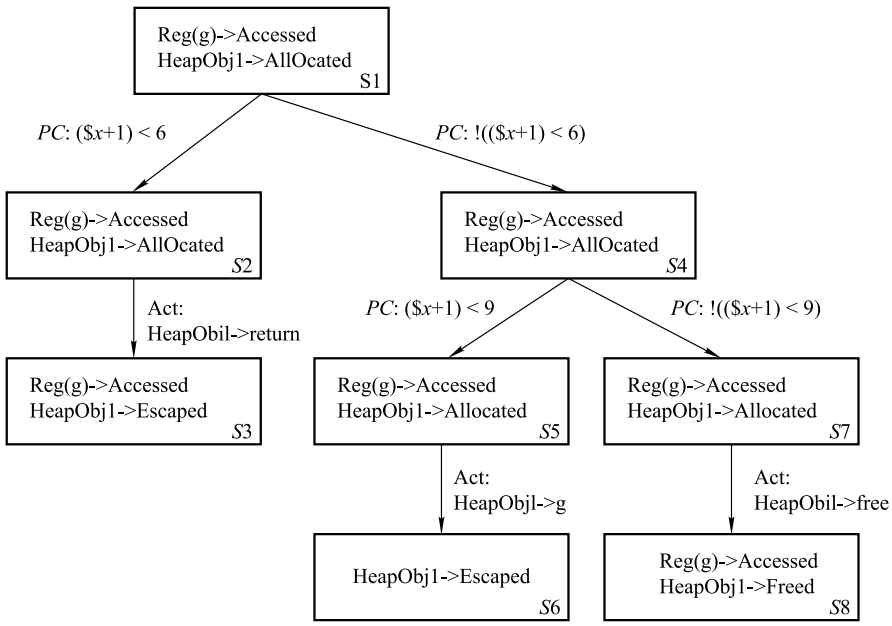

Fig. 14 The MSTG of function foo

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 constraints from the start node to a leaf node, i.e., one path in the program. The *UActs* is extracted from start node to leaf node. 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 are retained according to the following rules:

1) If the terminal state of a heap object is *Freed* or *Relinquished*, the actions referring to this object will be discarded.
2) If the terminal 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 terminal state of a heap object is *Escaped*, the actions *ToExtern* and *ToReturn* will be retained.
4) If the terminal state of an external object is *Freed*, the *ToFree* action will be retained.
5) If the terminal state of an external object is *Relinquished*, the *ToUnknown* action will be retained.
6) If the terminal 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. Algorithm 3 shows the algorithm of summary generation.

Algorithm 3 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 the rule 1 ~ 6
5: Add $\langle P, UActs\rangle$ to *Sum*
6: **end for**
7: **return** *Sum*

4.3.2 Summary application

A summary is a union of several $\langle P, UActs\rangle$ pairs. Algorithm 4 introduces the algorithm of summary application and the following steps describe the details.

1) Map the actual parameters to the formal parameters using access paths in line 1 of Algorithm 4. Access paths were first used by Landi and Ryder [25] 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, satisfiability is easier to reason about and feasible parts in summaries are probably reduced.
3) Apply the actions *UActs* to the caller's context. Each action can be represented by $m \rightarrow r$. The left side m that denotes a heap object or an external object should be transformed to the actual memory object first in the caller's context. An external object can be mapped to the actual memory object using access paths. A heap object is treated as the caller allocating a new heap object. We use *am* to denote the actual memory object. The right side r may denote a return value, a free symbol, an unknown symbol, or an external pointer. The process of each kind is described in Algorithm 4. Take the following code as an example:

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

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

Algorithm 4 The algorithm of summary application

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: **for all** *Act* in *UActs* **do**
5: Get the memory object *m* that *Act* belongs to
6: Map *m* to the actual memory object *am*
7: **if** *Act* is *ToExtern* with the external pointer *ep* **then**
8: Map *ep* to the actual pointer *ap*
9: Construct the new *NAct* from *am* and *ap*
10: *N* = *UpdateState*(*N*, *am*, *NAct*)
11: **else if** *Act* is *ToFree* **then**
12: *N* = *UpdateState*(*N*, *am*, *ToFree*)
13: **else if** *Act* is *ToUnknown* or *ToAlloc* **then**
14: *N* = *UpdateState*(*N*, *am*, *ToUnknown*)
15: **else if** *Act* is *ToReturn* **then**
16: Set the right value of the calling expression as *am*
17: **end if**
18: **end for**
19: **end if**
10: **end for**

4.4 An illustrating example

We use Fig. 7 to illustrate our leak detection algorithm. For

convenience, we list it again below:

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

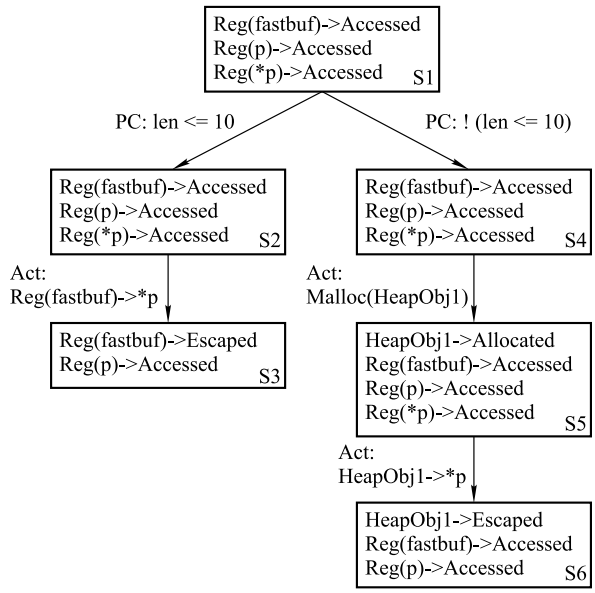

Fig. 15 MSTG of function *bufSelect*

As stated before, some tools miss leaks in this example due to a lack of global pointer analysis and precise function behavior model. Others report false positives because path conditions are discarded during the intra- or inter-procedural analysis. Using a precise MSTG model, our approach can identify the memory leak in Fig. 7 and the false positive can be eliminated.

First, we analyze the function `bufSelect` as an MSTG shown in Fig. 15. At the start, the external objects $Reg(p)$, $Reg(*p)$, $Reg(fastbuf)$ passed in as function parameters are initialized to the *Accessed* state. Then we compute the feasibility of the first path condition (PC) `len<=10`. If it is feasible, the PC is annotated on the edge $S1 \rightarrow S2$ without changing any state. Then we analyze the statement, i.e., `*p = fastbuf` in this branch. The action of assigning $Reg(fastbuf)$ to pointer `*p` is recorded on the edge $S2 \rightarrow S3$ causing the state of $Reg(fastbuf)$ to become *Escaped*. At the same time $Reg(*p)$ is removed as no pointer points to it in this function. Another branch with PC `!(len<=10)` annotated on edge $S1 \rightarrow S4$ allocates a new heap object *HeapObj1* in transition $S4 \rightarrow S5$, which is assigned to pointer `*p` in transition $S5 \rightarrow S6$. If leaked states exist in nodes, leaks will be reported.

After the analysis of the function `bufSelect` is finished, we generate a function summary for `bufSelect` from the MSTG. Each path in MSTG is extracted as a part of the summary called function effect. Each function effect consists of the annotated path conditions and the actions. 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 condition and *UActs* is a set of actions collected from one path in MSTG. P is the conjunction of the branch constraints annotated on edges in one path of the MSTG. *UActs* is a merged set of actions, some of which may be discarded and others may be retained. In this example, most of the actions are retained except for $Malloc(HeapObj1)$. This action is implicitly represented in *HeapObj1*, indicating that if we have actions on heap objects, these heap objects always have the *Malloc* action before other actions.

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 `buf`. 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 is reported because the second branch condition `!(len<=10)` is infeasible. Since 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`.

4.5 The advantages and potential optimizations

As stated before, the MSTG model is an abstraction of the program that stores the relevant information about memory allocation and memory release. Compared with other similar models, from memory abstraction to model building, MSTG stores more detail of the memory changes (recall the compar-

ison to related works in Section 2). We classify such changes into more categories, so as to deal with different situations. With such a model, we can perform a more accurate analysis.

The MSTG model can also provide some potential optimizations during the analysis. One such optimization is duplicate state removal. When creating a new memory state by evaluating a statement, the algorithm checks if this new state, at the current program point, has been generated previously by hashing three objects: the memory content of this new state, the current program point, and labels along path from starting node to current node in MSTG. If it has been generated previously, we stop exploring this new state since it has been done before. For example, given the following code snippet,

```
1 p = heapobj;
2 if (*)
3    x = p;
4 else
5    x = heapobj;
6 ...
```

where * denotes a complex constraint that cannot be handled, the two paths $1 \to 2 \to 3 \to 6$ and $1 \to 2 \to 5 \to 6$ have same effect at line 6. So the states of these two paths are merged at line 6 without losing any precision.

Another potential optimization is summary simplification. Generally, a single path is extracted as one part of a function summary, i.e. a tuple {predicate set, memory action set}. For those parts with the same memory action set, we can merge them by the union of their predicate sets into one part. This reduces the number parts of a function summary, which may improve the performance of summary application. We leave the implementation and investigation of this as a future work.

5 Implementations

We implemented our tool called **Melton** in Clang [26], a C language family frontend for LLVM compiler infrastructure. Clang provides a powerful intraprocedural framework for static analysis, built as clang static analyzer (CSA) [27]. We extended it to support interprocedural analysis for memory leak detection by adding a memory leak checker and a function summary generator. We also improved the bug reporter to provide richer diagnostic information.

5.1 Clang static analyzer overview

CSA is a source code analysis tool that finds bugs in C and Objective-C programs. It performs intraprocedural path-sensitive analysis to detect several kinds of bug. It provides a precise symbolic execution engine and a simple constraint solver.

5.1.1 Symbolic execution engine

The engine tries to evaluate all expressions with symbolic values. The region-based memory model [11] is used to statically simulate memory states at each program point. It abstracts C programs by introducing several kinds of regions with hierarchy that can handle almost all kinds of C expression, such as arbitrary levels of pointer dereferences, pointer arithmetic, and composite array. Since it provides precise pointer manipulation, it is suitable for detecting memory leaks.

5.1.2 Constraint solver

The solver simply performs interval analysis. For pointer variables, it only tracks whether a pointer can be `NULL` or not. For integer variables, it records the interval for each single symbol. The implementation and solving ability are not the focus of this article. Currently, the satisfiability checking of path constraints in Melton is performed using this solver.

5.2 Memory leak checker

The memory leak checker interacts with the symbolic execution engine. The memory objects are created in the memory leak checker and are passed to symbolic execution engine to track object values and aliases. MSTGs are built by the checker when it evaluats each statement. Function summaries are generated after the analyses of the functions are finished.

5.3 Bug reporter

A main challenge in practical application of static tools is the high manual cost to confirm bug reports especially when the false positive rate is high. In addition, a memory leak is one of the hardest bugs to detect since it has few visible symptoms other than the slow and steady increase in memory consumption. A helpful bug reporter is needed to provide rich diagnostic information and guidance for confirming and correcting memory leaks.

The bug reporter implemented in CSA annotates the program path segments along which bugs are triggered. However, the bug reporter is incapable of identifying the reasons of leakage and is difficult to check if a bug report is sound. Besides annotating leakage path segments, our bug reporter

classifies leakage into two types and identifies not only allocation sites but also the call trace of allocation sites. We list the features of Melton here:

- Leakage path segments are annotated in source code displayed with HTML files. Since our analysis is path-sensitive, a large number of leakage paths may be given from start to the leakage program point. So Melton provides an option "hash-reports=level" with user-specified levels 0, 1, 2, to control the number of reports. Level 0 represents outputting all of the paths that contribute to memory leaks. Level 1 uses the allocation site and leakage program point as key values to filter reports with the same key. Level 2 only uses the allocation site as the key value.
- Calling traces of allocation sites are underlined with hyper links, following which we can figure out where the heap objects are allocated and how they escape (from parameters, return value, or others).
- Memory leaks are classified into two types: lost memory and forgotten memory. Lost memory refers to the situation where a heap object becomes unreachable without first being deallocated. For example, in `p = malloc(...); p = NULL;`, the object `malloc(...)` loses all references after `p` is assigned to `NULL`. Forgotten memory refers to the situation where a heap object remains reachable but is not deallocated or does not escape when exiting the function [24]. For example, given `void f() { int p =malloc(...); }`, the object `malloc(...)` is only refereed by local pointer `p`. It gives the programmers an intuitive reason how memory leak occurs in the source-code level.

Figure 16 shows a bug report from `OpenSSH`. A heap object is allocated at the function call `buffer_init(&queue)`. It takes one true branch, one false branch and then reaches a return statement. Following the hyper-links starting from `buffer_init(&queue)`, we can identify the allocation trace.

6 Experimental results

In order to investigate the capabilities of our approach for detecting memory leaks, we applied Melton to five open source programs and some programs from SPEC 2000. The five open source programs are two libraries `sqlite` and `libosip` that are usually used in long running systems, an SSH connectivity tool `OpenSSH` and two packages from GNU. All of these programs are real-world code and more than 90% of leaks were new and have been confirmed and fixed by their maintainers. We compare our tool with CSA, LC, Clouseau, Saturn, FastCheck, SPARROW and Saber in Section 6.2. The platform we used for experiments is an Intel Xeon processor, 3.2GHZ, with 32G RAM.

6.1 Leaks found

6.1.1 Leaks in open source projects

Melton was applied to five open source projects and the results are given in Table 4. The first column shows the names of the programs. The second and third columns describe the

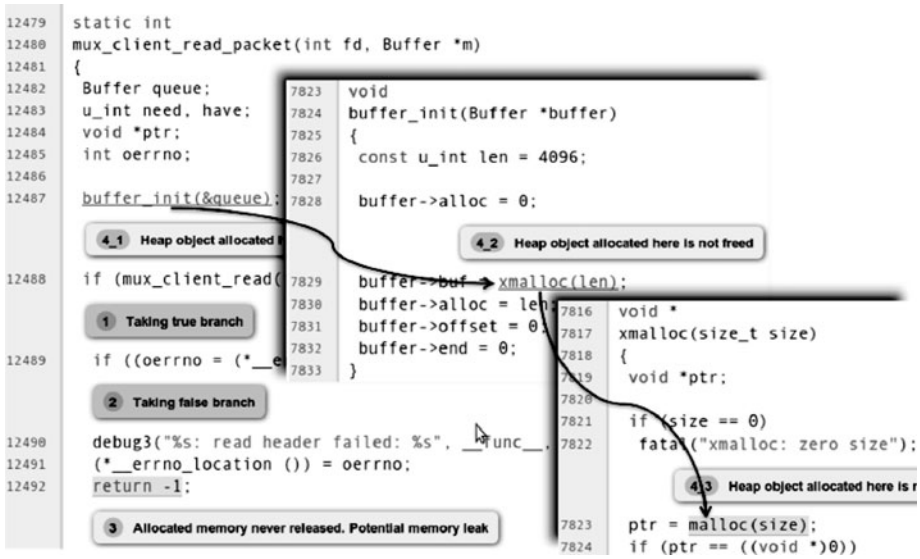

Fig. 16 An example bug report

number of lines of each program in thousands of lines and analysis time in seconds. The following two columns show the real bugs with lost memory type and forgotten memory type. The final column gives the total number of real bugs.

Table 4 Experimental results on selected GNU open source projects

Programs	Size /kl	Time /s	Lost	Forg.	Total
which-2.16	2.8	31	0	1	1
libosip2-3.6.0	29.0	16	10	11	21
wget-1.13.3	91.8	522	2	8	10
openssh-5.9p1	89.8	383	1	6	7
sqlite-3.7.11	139.2	54 752	0	2	2

Melton found 41 bugs in these five programs. Many bugs in `libosip` come from mishandling of list insertion. When the program tries to insert a newly allocated node to the list and the size of the list has reached the maximum value, the node will not be inserted into the list and without any deallocation action, leak occurs. The bugs in `wget` are mainly caused by the forgotten deallocation on string buffers. The bugs in `OpenSSH` are from `ssh` utility. Note that each reported bug derives from different root causes, which means if a leak occurs in a function, Melton only generates one bug report no matter how many times this function is invoked.

Figure 17 shows a memory leak found by Melton in `OpenSSH`. At line *M*, `match_list` allocates a heap object to `name` which is not freed at the second loop iteration before reassigned to `name`. In fact, this is a clerical error as the fix is just moving `if` branch at line *F* into the end of the `for` loop. It belongs to lost memory type.

```
   Authmethod *authmethod_get (…)
     …
     for (;;) {
M:     if (name = match_list (…) == NULL) {
         return NULL;
       }
       if (…) {
         xfree (name);
         return …;
       }
     } //end for
F:   if (name != NULL)
       xfree(name);
   }
```

Fig. 17 Memory leak in file sshconnect2.c in OpenSSH-5.9p1

Figure 18 shows a forgotten memory leak found in `wget`. The heap object is created at line *M* by `request_new` and is forgotten to be deallocated at line *L*. It takes 47 different branches to trigger this bug.

All of the programs are the latest version at the time we analyzed. More than 90% of leaks are newly found and reported by Melton. Other leaks in `wget` have been fixed (are not new) before we reported them. But the reports are accurate. The leak reports can be found on the Melton homepage[2]. These results indicate that Melton is more practical on analyzing real-world programs than other tools. As far as we know, the reports in the experimental results of other tools either are only checked manually or derived from old benchmarks.

```
   static uerr_t gethttp (…) {
M: req = request_new ();
     if (…)
       …
     else if (…)
       …
     else {
       request_free (req);
       return …;
     }
     if (…) {
       if (…) {
L:       return …;
       }
     }
   }
```

Fig. 18 Memory leak in file http.c in wget-1.13

6.1.2 Leaks in SPEC CPU 2000

Table 5 shows the experimental results. The first column shows the names of the programs. The second and third columns describe the number of lines of each program and analysis time. The subsequent two columns show the number of real leaks and false positives Melton reported. Three programs `gap`, `perlbmk`, and `vpr` are not compared as the front-end Clang could not process them.

Table 5 Experimental results on some programs of SPEC CPU 2000

Prog.	Size /kl	Time /s	Leaks	FP
ammp	13.3	71.3	20	0
art	1.3	2.9	0	0
bzip2	4.6	7.0	0	1
crafty	18.9	631.8	0	0
equake	1.5	0.2	0	0
gap	59.5	N/A	N/A	N/A
gcc	205.8	50 688	39	3
gzip	7.8	55	0	0
mcf	1.9	1	0	0
mesa	49.7	1 300	3	10
parser	10.9	223	0	0
perlbmk	58.2	N/A	N/A	N/A
twolf	19.7	340	0	0
vortex	52.7	27 469	559	0
vpr	17	N/A	N/A	N/A

[2] http://lcs.ios.ac.cn/~zj/pa.html

Twenty real leaks are detected by Melton in `ammp`. All of these leaks are caused by the same code pattern:

```
a = malloc(len);
if (a == NULL) return;
b = malloc(len);
if (b == NULL) return;
```

The leakage of the first allocated object occurs if the second allocation fails. This kind of leak pattern also exists in `mesa`. Another kind of memory leak in `mesa` comes from the mishandling of abnormal cases. Heap space is created for storing an image file and if the type of this image is a bad one, the program just returns NULL without deallocating the heap space.

Most leaks in `gcc` come from mishandling of the concatenation of strings. The function call `concat(a, b)` returns a newly allocated string concating `a` with `b` without changing `a` and `b`. A leak occurs when function `concat` is called in this way: `concat(a, concat(b, c))`; in which the heap object allocated by `concat(b, c)` leaks.

For `vortex`, Melton reported 559 leaks. After checking the code in `vortex`, we found that each allocated object is assigned to a global pointer:

```
void * VoidLowestAddr;
addrtype Void_ExtendCore(...) {
  //Allocated here
  LongPtr = calloc(1, FarSize);
  CoreAddr = LongPtr;
  if (CoreAddr < VoidLowestAddr)
    // Assigned to global value
    VoidLowestAddr = CoreAddr;
  ...
  return CoreAddr;
}
```

Each time function `Void_ExtendCore` is called, the caller does not release the returned heap object. If the value of `CoreAddr` in the later calls is lower than the previous addresses, the previously allocated heap object leaks.

Compared with the experimental results in our conference paper [28], Melton detected more leaks (about 558) in `ammp` and `vortex` and eliminated some false positives in `mesa` and `vortex`. The improvement is mainly introduced by the refinement of our memory abstraction implementation.

6.2 Comparison with other tools

In the following sections, we compare Melton with the CSA v3.0 on the programs analyzed in the previous section and with other tools on overall false negative and false positive.

6.2.1 Comparison with CSA

The CSA implements several checkers for finding bugs. However, the lack of interprocedural analysis limits its capabilities. Since Melton focuses on memory leaks, we compare the built-in memory leak checker in CSA with our tool. Table 6 shows the experimental results by giving analysis time (Time), the number of real leaks (Leaks) and false positives (FP). CSA spends less analysis time but does not find any error in these programs. For false negatives, as most of heap objects are allocated via several function calls, CSA is not able to identify the allocation action of user-written functions without interprocedural analysis and missed the bugs Melton reported. For false positives, CSA cannot model the free action or escape action in user-written functions and generates lots of false positives. Melton reported 41 real bugs with 31 false positives and has better capabilities than CSA.

Table 6 Comparison with Clang

Programs	Melton			CSA		
	Time /s	Leaks	FP	Time /s	Leaks	FP
which-2.16	31	1	0	6	0	0
libosip2-3.6.0	16	21	2	2	0	84
wget-1.13.3	522	10	12	62	0	12
openssh-5.9p1	383	7	15	47	0	82
sqlite-3.7.11	54 752	2	2	58	0	3

6.2.2 Comparison on False Negative

Table 7 presents findable leaks only by path-sensitive interprocedural analysis (PSIPA) and Melton based on previous experimental results. The first column gives the analyzed programs, in which **5OSP** represents the five open source programs. The second column shows the total bugs found in these programs by Melton. The third and forth column respectively present the number of bugs only found by path-sensitive interprocedural analysis and Melton.

Table 7 Findable bugs by path-sensitive interprocedural analysis or Melton

Programs	Total	PSIPA	Melton
5OSP	41	21	1
SPEC CPU 2000	621	0	559

There are 21 bugs that can be found only when performing path-sensitive interprocedural analysis. This means, in tools mentioned above, only Saturn and Melton can find these bugs. Take the following code (from `libosip`) as an example,

```
evt = __osip_ict_need_timer_a_event(...);
if (evt != ((void *)0))
 osip_fifo_add(tr->transactionff, evt);
```

in which `__osip_ict_need_timer_a_event` creates a new heap object in `evt`, and then function `osip_fifo_add` tries to insert `evt` into a global queue. This insertion succeeds when the queue has not reached the max length, otherwise, it just returns `OSIP_FAILURE`. For tools without full path-sensitivity, the summary of function `osip_fifo_add` is merged as `evt` is always added to a global queue. Thus, if `evt` is not freed, without full path-sensitivity, leaks will be missed.

Besides, as Melton captures more precise memory actions, there are 560 (1 + 559) leaks that are only findable by Melton. Most of these leaks involve the *ToGlobal* action. The following code in `wget` has a leak,

```
char **pstring;
bool setval_internal_tilde(...) {
   ...
   if (...)
      home = home_dir();
   //Allocation Site
   *pstring = concat_strings(home, ...);
}
bool run_wgetrc(char *file) {
   ...
   while((line= read_whole_line(file)!=NULL)){
      ...
      switch (...)
      case ...:
         // May be called twice
         setval_internal_tilde(...);
      ...
   }
}
```

in which function `setval_internal_tilde` allocates a heap object to the global variable `*pstring`, and when this function is called twice in function `run_wgetrc`, the heap object allocated the first time leaks. Since other tools cannot capture the actions including global variables, they miss this kind of leaks.

Although some extra leaks are found by our tool, after checking other tools' bug reports, we find that Melton also misses some bugs due to two limitations in our tool. The first limitation is the unsound handling of loops. Since programs may contain a large number of paths, especially for programs with several loops, we bound the number of loop iterations. Program paths outside the loop bound will not be analyzed and leaks on these paths will be missed. It causes the false negatives in `art`, `bzip2` and `twolf`. For example, consider the following code:

```
char *array = calloc(len);
for (i = 0; i < 1024; i ++) {
   if (...)
   else ...
}
```

The `for` loop iterates 1024 times. In Melton the default maximum iteration number is 2 and the detection process will be stopped after the second iteration. So we are unable to check if the heap object that the `array` points to leaks.

The second limitation is the weakness of handling the *ToAlloc* action when performing interprocedural analysis. As shown in the following code,

```
list *new_and_next() {
   list *l = (list *)malloc(sizeof(list));
   l->next = (list *)malloc(sizeof(list));
   return l;
}
void foo() {
   list *l = new_and_next();
   ...
   free(l);
}
```

in which the heap object allocated to `l->next` is relinquished by Melton and leaks are missed if `l->next` is not deallocated. This causes the false negatives in `mesa`.

6.2.3 Comparison on false positives

Table 8 the presents false positive ratio of other tools and Melton. The first column gives the tool names and the second column shows the size of the analyzed programs in thousands of lines. The **Leak Count** column gives the number of real leaks found by each tool and the **FP Ratio** column shows the false positive ratio of each tool. This table shows our tool reports the most number of real leaks with the lowest false positive rate.[3)]

The causes of false positives in our tool can be classified into the following reasons:

- Function pointers are hard to process, especially for those in parameters and global variables. Since our approach is summary-based analysis, values of function pointers cannot be retrieved in the current context. If a function pointer points to a *free* function while Melton misses it, it may cause a false positive. Most of the false

[3)] Note that the result on false positive rate should be regarded only as rough estimates, since these tools are applied to different benchmarks. We do not compare them with the same benchmarks because some of them are no longer available and even for available ones, many of the compiler front-ends are too old to accept the benchmarks we analyze such as the five open source projects mentioned above.

Table 8 Comparison on False Positive

Tools	C size /kl	Leak count	FP ratio /%
LC	321	26	56
Clouseau	1 086	409	64
Saturn	6 822	455	10
FastCheck	671	63	14
SPARROW	1 777	332	12
Saber	2 324	211	19
Melton	740.7	662	6

positives in `mesa` come from this problem.

- The weak handling in pointer arithmetic also raises some false positives. For instance, if `p` points to a heap object and then the program executes `p++`. Our tool treats it as a leakage since no pointer points to the heap object. However, we can access the heap object via `p-1`. This causes one false positive in `bzip2`.
- Currently, constraints are solved by using the solver implemented in CSA that only supports interval analysis. It can be improved by adding and external SMT solver such as STP [29].
- The external library calls without source code cannot be modeled accurately. This raises three false positives in `OpenSSH`.

7 Conclusion and future work

We have presented a novel analysis model called MSTG for detecting memory leaks. Our model captures procedure behaviors more precisely and reasons about path conditions to reduce the false positive rate and path explosion. To support interprocedural analysis, it generates precise summaries that are applied at function's call sites. Several memory leaks have been found in real programs and more leaks have been detected in some programs of SPEC CPU 2000 benchmarks.

Currently, Melton misses some bugs due to the weak handling of loops, recursions, and recursive data structures. In future work, we plan to optimize in loops with constant iteration number and to capture the memory actions in recursive data structures. In addition, we will improve our memory abstraction model to describe function side-effects more precisely and further reduce the false positive rate.

Acknowledgements This work was partially supported by the 973 Program of China (2014CB340701) and the National Natural Science Foundation of China (Grant No. 61003026).

References

1. Evans D. Static detection of dynamic memory errors. In: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design an Implementation. 1996, 44–53
2. Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors. Software-Practice and Experience, 2000, 30(7): 775–802
3. Heine D L, Lam M S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. 2003, 168–181
4. Xie Y, Aiken A. Context- and path-sensitive memory leak detection. In: Proceedings of the 2005 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2005, 115–125
5. Orlovich M, Rugina R. Memory leak analysis by contradiction. In: International Static Analysis Symposium. 2006, 405–424
6. Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation. 2007, 480–491
7. Xu Z, Zhang J. Path and context sensitive inter-procedural memory leak detection. In: Proceedings of the 2008 International Conference on Quality Software. 2008, 412–420
8. Jung Y, Yi K. Practical memory leak detector based on parameterized procedural summaries. In: Proceedings of the 2008 International Symposium on Memory Management. 2008, 131–140
9. Wang J, Ma X D, Dong W, Xu H F, Liu W W. Demand-driven memory leak detection based on flow and context-sensitive pointer analysis. Journal of Computer Science and Technology, 2009, 347–356
10. Sui Y, Ye D, Xue J. Static memory leak detection using full-sparse value-flow analysis. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. 2012, 254–264
11. Xu Z, Kremenek T, Zhang J. A memory model for static analysis of C programs. In: Proceedings of the 2010 International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. 2010, 535–548
12. Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. In: Proceedings of the Winter USENIX Conference. 1992, 125–138
13. Mitchell N, Sevitsky G. Leakbot: an automated and lightweight tool for diagnosing memory leaks in large java applications. In: Proceedings of the 2003 European Conference on Object-Oriented Programming. 2003, 351–377
14. Hauswirth M, Chilimbi T M. Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. 2004, 156–164
15. Bond M D, McKinley K S. Bell: bit-encoding online memory leak detection. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. 2006, 61–72
16. Jump M, McKinley K S. Cork: dynamic memory leak detection

for garbage-collected languages. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2007, 31–38

17. Xu G, Rountev A. Precise memory leak detection for java software using container profiling. In: Proceedings of the 2008 International Conference on Software Engineering. 2008, 151–160
18. Distefano D, Filipović I. Memory leaks detection in java by bi-abductive inference. In: Proceedings of Fundamental Approaches to Software Engineering. 2010, 278–292
19. Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. 2005, 213–223
20. Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for c. In: Proceedings of the 2005 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2005, 263–272
21. Cadar C, Ganesh V, Pawlowski P M, Dill D L, Engler D R. EXE: Automatically generating inputs of death. In: Proceedings of the 2006 Conference on Computer and Communications Security. 2006, 322–335
22. Xie Y, Aiken A. Scalable error detection using Boolean satisfiability. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 2005, 351–363
23. Kildall G A. A unified approach to global program optimization. In: Proceedings of Principles of Programming Languages. 1973, 194–206
24. Clause J, Orso A. Leakpoint: pinpointing the causes of memory leaks. In: Proceedings of the 2010 International Conference on Software Engineering. 2010, 515–524
25. Landi W, Ryder B G. A safe approximate algorithm for interprocedural aliasing. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. 1992, 235–248
26. Clang: a C language family frontend for LLVM. http://clang.llvm.org
27. Clang static analyzer. http://clang-analyzer.llvm.org
28. Xu Z, Zhang J, Xu Z. Memory leak detection based on memory state transition graph. In: Proceedings of the 2011 Asia-Pacific Software Engineering Conference. 2011, 33–40
29. Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays. Lecture Notes in Computer Science, 2007, 4590, 519–531

Zhenbo Xu received his BS in the Department of Computer Science and Technology, University of Science and Technology of China (USTC) in 2009. He is currently a PhD candidate in USTC. He has been visiting the Institute of Software, Chinese Academy of Sciences since 2010. His research interests include static analysis and bug finding.

Jian Zhang received his PhD from the Institute of Software, Chinese Academy of Sciences (ISCAS) in 1994. He is a professor and assistant director of ISCAS. He is a senior member of ACM, IEEE and CCF. His research interests include finite model searching, satisfiability checking, program analysis and software testing. He has served on the PC of more than 60 international conferences.

Zhongxing Xu received his PhD from the Institute of Software, Chinese Academy of Sciences (ISCAS) in 2009. He was an assitant professor of ISCAS during 2009–2012. His research interests include static analysis and bug finding. Now he is a software engineer in a software company.