A Projection-based Approach for Memory Leak Detection

***Abstract*—One of the major software safety issues is memory leak. Moreover, detecting memory leak vulnerabilities is challenging in static analysis. Existing static detection tools find bugs by collecting programs' information in the process of scanning source code. However, the current detection tools are weak in efficiency and accuracy, especially when the targeted program contains complex branches. This paper proposes a projection-based approach to detect memory leaks in C source code with complex control flows. According to the features of memory allocation and deallocation in C source code, this approach projects the original control flow graph of a program to a simpler one, and it reduces the analysis complexity. Besides, this paper implements a memory-leak detection tool—PML_Checker[1], and evaluats the tool by comparing with three open-source static detection tools on both public benchmarks and study test cases. The experimental results show that PML_Checker reports the most memory leak vulnerabilities among the four existing tools in C source code with complex control flows and complex data types, and PML_Checker obtains higher efficiency and accuracy on public benchmarks.**

I. Introduction

Memory leak is a major cause of reliability and performance issues in software. Especially in embedded systems, since it causes the long-running applications, which eventually will run out of memory. A system running out of memory may lead to the slowing down of the system caused by frequently swapping in and out, and process creation failure because of no more memory available. Moreover, memory leaks have caused severe consequences in some large applications and services. For example, on October 22, 2012, in the US-East region, a potential memory leak caused Amazon Elastic Block Store (EBS) and Elastic Compute Cloud (EC2) services to be six hours partially disrupted, and during this time services can not handle IO requests[2]. Therefore, detecting memory leaks is very important and necessary to ensure the quality of software. Although memory leaks do not typically constitute a direct security threat, attackers can exploit them to increase a denial-of-service attacks effectiveness. However, detecting memory leaks is challenging, since the only symptom of memory leaks is the slow increasing of memory consumption.

To address this challenge, there are two general techniques in existing works—dynamic testing and static source code analysis [1]. Dynamic testing relies on the coverage of the test cases and requires long-time execution. Comparing to dynamic testing, static source code analysis has the advantage of higher accuracy, because static analysis is usually to find the memory allocation location and the corresponding release point. Therefore, this paper focuses on detecting memory leaks by applying source code analysis. Among source codes written in various languages, detecting memory leaks is particularly necessary when the program is written in low-level programming languages such as C/C++. Because low-level programming languages allow manual memory management, such as explicit memory allocation and deallocation [2]. Such manual memory operations are often error-prone, and even lead to vulnerabilities.

The relationship between memory allocation and deallocation can describe the root causes of memory leaks. In other words, the following two points show the classification of memory-leak reasons [3](a)The dynamic memory blocks are not free/deallocated; (b)The dynamic memory blocks are freed in wrong order. To check the above two errors, there are in general the following two approaches: value-flow-based approach and control-flow-based approach. The basic idea of value-flow-based approach is to capture def-use chains and value flows, by assigning all memory locations represented by both top-level and address-taken pointers. Thence, it is also known as VFG (Value Flow Graph). While the basic idea of control-flow-based approach is to construct a dynamic memory allocation and deallocation model on the CFG (Control Flow Graph). Based on the CFG model, this paper checks whether a block of heap memory space is reclaimed by the program when the lifetime of the program has ended—if a block is not reclaimed, then there is memory leak. Comparing the VFG model [4] with the projection-based CFG model, the latter focuses on all the possible execution paths, analyzes the lifetime of pointers that assigned memory locations. In particular, in order to capture the indirect allocation-deallocation relation and obtain accurate results.this paper takes into account the memory pointers' lifecycle, i.e., including all the memory operations.

The existing control-flow-based approaches have limitations in efficiency and accuracy when the control flow is getting complex, because the analysis of complex control flow needs to deal with a large number of path branches. We say that the control flow of a program is complex if the allocation and deallocation appear in different control flow branches of a program, and thus memory leaks are more likely to happen. Due to the complexity of the branch conditions' analysis, complex control flows make memory detection more difficult. In this work, this paper focuses on efficiently and accurately detecting memory leaks in programs with complex control flows.

The higher complexity of the control flows is, the easier

[1]PML_Checker.https://github.com/sunxiaohuiczcz/PML_Checker.git.2017.

[2]AWS Service Event.https://aws.amazon.com/cn/message/680342/.2017.

the vulnerabilities can hide. To detect memory leaks, detection system needs to search each control flow. Thus, the difficulty of detecting memory leaks in complex control flows increases with the complexity of paths in the control flows. Properly reducing the complexity of programs control flows can greatly improve the efficiency and accuracy of detecting memory leaks in the program. Therefore, in this work, this paper proposes to apply projection to the control flow grap. The projection abstracts the original control flow graph to a simplified graph to reduce the complexity of searching control flows in memory detection. The main challenge in the abstraction is to ignore the irrelevant paths. To address this challenge, when program has complex control flow branches, detecting system needs to evaluate the condition value of each control flow branch node, and then analyze the execution results of different branch paths according to different conditions. In order to achieve this goal, our system adopts path-sensitive analysis method [5]. This method records different program states of complex branch paths by tracing each branch of the program. This method analyzes the combinatorial relationship between branches. Thus the path-sensitive analysis method identify all the paths in a control flow graph. In other words, this method will abstract away some paths that are irrelevant to memory operations by matching the defined expression. A key issue in path-sensitive analysis is the evaluation of the guiding conditions of paths. To address this issue, our system uses a tool named PAT (Path Analysis and Testing) [6]. PAT uses symbolic execution and constraint solving, to perform accurate evaluation on branch evaluations. As a result, this paper proposes a set of rules for guiding the projection. Our approch essentially reduces irrelevant nodes in a control flow graph, and proves the correctness of the rules—no irrelevant path is considered and no relevant path is ignored.

The main contributions of this paper can be summarized as follows:

- The classification of memory leaks, particularly in source code with complex control flows.
- A projection-based approach to detect potential memory leaks in C program, reducing the false negative rate in the program with complex control flows.
- A detecting tool—PML_Checker, evaluating the effectiveness and accuracy of the tool on public benchmarks (SPEC CPU 2000, SIR and SARD).

In the next section this paper introduce our control flow graphs projection approach and prove its effectiveness. Section II analyzes the causes of memory leaks and describe a leak model from the perspective of path-abstraction. Section III mainly presents the projection algorithm and memory leaks detection algorithm. In Section IV and Section V, we conduct experiments to test PML_Checker by comparing with three open-source detection tools, and we evaluate it at its effectiveness, accuracy, efficiency and scalability. Section VI discusses related research. And finally we conclude this work in Section VII.

II. Problem Description

The targeted problem in this paper is the dynamic memory leaks—A block of heap memory space is being leaked if the program, or the run-time system does not reclaim its memory when the lifetime of heap memory space has ended. The lifetime of heap memory space is represented in three ways [7]:

- *Referencing:* the lifetime of a block of a heap memory space ends when there are no references to the space (excluding references from abandoned spaces).
- *Reachability:* the lifetime of a heap memory space ends when it is no longer reachable from program variables.
- *Liveness:* the lifetime of a heap memory space ends after the last access to that space.

This paper focuses on the second view—reachability. Reachability is more intuitive in the CFG, and our analysis is based on it. In the reachability view, there is a memory leak if the lifetime of a heap memory space ends while the memory pointer can not reach to that space. Since our approach is based on CFG, detecting memory leaks is essentially analyzing whether the memory pointers have been freed when the lifetime of the corresponding memory spaces end in each branch of the control flow graph. Therefore, the memory detection can be converted into the control flow graph analysis. The analysis includes the correspondence relations between the memory pointers and the lifetime of the corresponding memory spaces. That refers to the reachability of the pointers.

As stated in Section I, when procedure P has complex control flow branches, the pointers pointing to corresponding memory spaces may be freed in wrong order, and the allocation and deallocation of memory space may appear in different control flow branches. Thus increases the complexity of the control flow graph analysis, because of the complex branch guarding condition analysis. This paper formally defines the complex control flow as follows.

Definition 1 (Complex Control Flow)**.** *Given that a directed graph named $G = (N, E)$ is the control flow graph of a procedure P, where N denotes the set of nodes and E denotes the set of edges, the control flow is called* complex control flow *when the cyclomatic complexity of G, denoted as $V(G)$, is no less than 2. That is $V(G) = D + 1 \geq 2$, where D denotes the number of conditional and loop branching nodes.*

Fig. 1 displays this generation of complex control flows. A simple control flow graph of procedure P with memory allocation and deallocation is shown on the left. After adding the control flow branches, a complex control flow graph is shown on the right. Therefore, detection system needs to analyze whether the memory spaces are released in each branch after being allocated. It should be noted that this section only considers the cases that allocate or release memory spaces of the same pointer, and the case that memory spaces are not fully released does not exist. There are four cases corresponding to the values of the two branches in the complex control flow graph:

1) B1 =$false$, B2 = $false$: Corresponding to path 1-6-7. No memory leaks occur in this case, since there is no memory operations.
2) B1 =$true$, B2 = $false$: Corresponding to path 1-2-3-7. The memory blocks have not been freed after being allocated, which is a typical kind of memory leaks.
3) B1 = $false$, B2 = $true$: Corresponding to path 6-4-5. The unallocated memory blocks are freed.
4) B1 = $true$, B2 = $true$: Corresponding to path 1-2-3-4-5. Whether there is memory leak in this case is uncertain. The result relies on the number of executions of the memory allocation or deallocation related statements. Specifically, if both B1 and B2 are conditional branches, the memory allocation and deallocation will be executed only once, then there are no memory leaks in this structure. If B1 or B2 is a loop node, then the memory allocation or deallocation for the same pointer p will be executed more than once in the loop. It will lead to a memory leak. E.g., if B1 is a loop node, then multiple memory spaces are allocated, each in a single execution of the loop; but B2 is a conditional node, and thus there is only one deallocation in the branch.

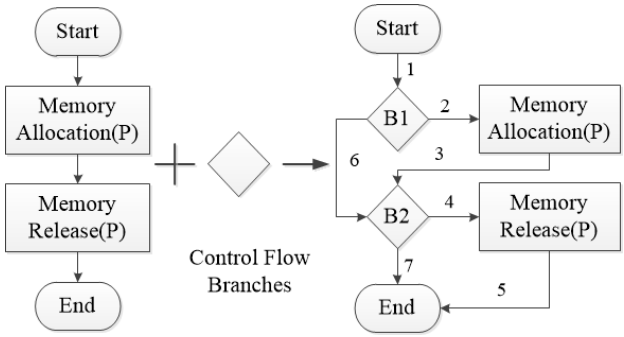

Figure 1. Description of memory leaks reasons

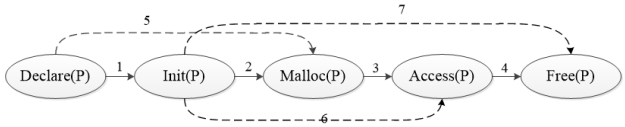

Figure 2. Lifetime of a pointer p

In each branch of the control flow graph, in addition to considering the allocation and deallocation of memory spaces, detection system needs to additionally consider the lifetime of each pointer, as shown in the definition of dynamic memory leaks. To provide some clue, Fig. 2 shows the lifetime of a pointer p pointing to a heap memory space, using C language as an example. This figure shows the lifetime of a pointer p from being declared to being freed. The solid arrows represent the correct order. The dotted arrows represent all the cases that may result in the memory leaks on the memory blocks refereed by a single pointer p. In details, the dotted arrow 5 shows the case that the pointer p is not initialized before being allocated to memory blocks, arrow 6 illustrates that the pointer p is not allocated to memory blocks before being accessed, arrow 7 presents that p is not allocated to memory blocks before being freed.

Note that Fig. 2 only shows the lifetime of one pointer. When considering multiple pointers, the cases that may lead to memory issues are more complicated. So that is another challenge in the memory leak detection.

III. Approach

Similar to the CFG based static analysis, this paper also first constructs a detection model of C program. The detection model contains different kinds of memory leaks. The main difference from our approach is that, our approach performs projection in constructing the detection model, and it simplifies memory leak detection procedure based on the model.

A. Projection Algorithm

This section describes a set of rules and an algorithm for projecting a control flow graph to a simply one.

Projection Subject is the input of the projection process and Projection Target is the output of the projection process. Specifically, Projection Subject(G) is the control flow graph of the program to be analyzed. $G = (N, E)$, where $N = \{n_1, n_2, \ldots, n_k\}$ denotes the set of nodes, and $E = \{e_1, e_2, \ldots, e_k\}$ denotes the set of edges. In the procedure P, each statement is a node n, and there is an edge e between two statements executed in sequence. Projection Target(G^*) is a directed graph that contains all the control flow branch nodes denoted by B, and all the allocation nodes denoted by A and deallocation nodes denoted by F (if exist). $G^* = (N^*, E^*)$, where $N^* = A \cup B \cup F$ ($A \cap B \cap F = \emptyset$) denotes the set of nodes, and E^* denotes the set of edges between nodes, representing the sequential ordering between nodes.

Definition 2 (Control Flow Graph Projection)**.** *Given two directed graphs $G = (N, E)$ and $G^* = (N^*, E^*)$. Let set M contains all the statement nodes relating to memory management in the procedure P and the set B contains all the control flow branch nodes (e.g.* if, else, while, for *et al.). If G and G^* satisfy the following conditions, then we say that G^* is the projection of G, denoted as $G^* \mapsto G$:*

- $N^* \subseteq N$;
- *If* $\forall m, n \in N^*, (m \neq n), if m \in \Gamma(n)$ *($\Gamma(n)$ denotes the set of direct successor nodes of n), then there must be a connected path μ from n to m in G, formally* $\exists \mu = (n, n_1, \ldots, n_l, m)$ *in G and* $n_1 \in \Gamma(n) \wedge n_{i+1} \in \Gamma(n_i) \wedge n_n \in \Gamma(m)$ *($0 \leq i \leq (n-1)$);*
- $\forall\ n \in B$ *in directed graph G^*, if* $\forall m \in M : m \notin \Gamma(n)$, *then* $\Gamma(n) = \Gamma(n) \cup \{n\}$.

In general, the Control Flow Graph Projection is the process from the Projection Subject transforming into the Projection Target according to the following rules. This paper summarizes the projection rules as follows:

- Rule 1: The set N^* of the projection graph only contains nodes related to memory allocation and deallocation, and all the control flow branch nodes in the procedure P.
- Rule 2: If a control flow branch contains the nodes related to memory allocation and deallocation, then the nodes that related to memory allocation and deallocation will be the successor nodes of this branch node in the projection graph G^*.
- Rule 3: If a control flow branch does not contain direct successor nodes that related to memory allocation and deallocation, then this branch node will form a closed node in the projection graph G^*. If the node $n \in N^*$, $n \in \Gamma(n)$, then the node n is a closed node.

Fig. 3 shows the CFG (G) of a piece of code. In this graph, there are 11 elements forms set N, that is, $N = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$. The memory block is allocated in S_1 ($S_1 \in A$) and freed in S_9 ($S_9 \in F$); S_3 is a looping branch node and S_6 is a conditional branch node ($B = \{S_3, S_6\}$); In node S_7, another pointer points to the memory block ($S_7 \in A$); Others are unrelated to memory operation. Thus the cyclomatic complexity of this graph is 4.

Fig. 4 shows the projection graph (G^*) of G. Following the projection Rule 2, we have $N* = A \cup B \cup F = \{S_1, S_3, S_6, S_7, S_9\}$. Following Rule 2, we have the edges from S_1 to S_3, S_3 to S_6, S_6 to S_7 and S_7 to S_9. Following Rule 3, S_3 is a closed node since all its direct successor nodes are not memory related nodes, thus we add a loop to S_3. There is no loop at S_6, because the direct successor s_7 is a memory related node.

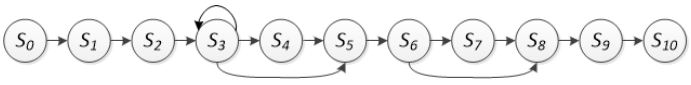

Figure 3. The CFG of the program (G)

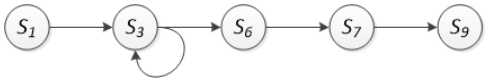

Figure 4. The projection graph (G*)

Theorem 1 (Control Flow Graph Projection Theorem)**.** *Given a directed graph $G = (N, E)$. If $G^* \mapsto G$ where $G^* = (N^*, E^*)$, then G^* strictly keeps the reachability relation among nodes in N^* from G, that is, $\forall m, n \in N^*$, m can reach n in G^* if and only if m can reach n in G.*

Proof: Presume m can reach n in G^*, which means there exists a directed path $\mu^* = (m, e_1^*, e_2^*, \ldots, e_k^*, n)$ in G^*. Since $G^* \mapsto G$, then there must exist a set of paths $\mu_0 = (m, e_i^0, \ldots, e_{j_0}^0, e_1^*)$, $\mu_i = (e_i^*, e_i^i, \ldots, e_{j_i}^i, e_{i+1}^*)$ $(1 \leq i \leq k-1)$ and $\mu_k = (e_k^*, e_i^k, \ldots, e_j^k, e_{j_k}, n)$ in G, where $e_{j_i}^i$ can be empty. The k paths are connected since the last node in a path is the starting node of another path. Formally, given path $\mu_i = (e_i^*, e_i^i, \ldots, e_{j_i}^i, e_{i+1}^*)$ and $\mu_{i+1} = (e_{i+1}^*, e_{i+1}^{i+1}, \ldots, e_{j_{i+1}}^{i+1}, e_{i+2}^*)$. According to the definition of path, we have $e_{i+1}^* \in \Gamma(e_{j_i}^i)$ in path μ_i and $e_{i+1}^{i+1} \in \Gamma(e_{i+1}^*)$ in path μ_{i+1}. Thus we have a path from $e_{j_i}^i$ to e_{i+1}^{i+1} via e_{i+1}^*. That is we can connect μ_i and μ_{i+1} as path $(e_i^*, e_i^i, \ldots, e_{j_i}^i, e_{i+1}^*, e_{i+1}^{i+1}, \ldots, e_{j_{i+1}}^{i+1}, e_{i+2}^*)$. In this way, by connecting the k paths, we have a path from m to n in G.

On the other hand, presume m can reach n in G, that is there exists a directed path $\mu = (m, \ldots, n)$ in G, then there must be:

- Either there are no other nodes in N^* except m and n;
- Or there is $\mu = (m, \ldots, n_1, \ldots, n_2, \ldots, n_k, \ldots, n)$, where m, n, $n_i \in N^* (1 \leq i \leq k)$, and the remaining nodes do not belong to N^*.

In the former case, according to the rules of control flow graph projection, m can immediately reach n in G^*, i.e., $n \in \Gamma(m)$. In the later case, let $\mu = m, \mu_1, n_1 \mu_2, n_2, \cdots, \mu_k, n_k, \mu_{k+1}, n$, where $\mu_i = (n_1^i, \ldots, n_{j_i}^i)(1 \leq i \leq k+1)$, there are no nodes in each path μ_i from N^* except end nodes. According to the projection rules, the nodes in each path μ_i will be deleted in G^* such that there is a immediate edge between m and n_1, between each n_i and $n_{i+1} (1 \leq i \leq k)$, and between n_{k+1} and n, which forms a path $\mu^* = (m, n_1, n_2, \ldots, n_i, n_{i+1}, \ldots, n_{k+1}, n)$ in G^*. ■

Algorithm 1 shows the projection algorithm according to the projection rules. Given the control flow graph $G = (N, E)$ of program P as input. The algorithm returns G's projection graph $G^* = (N^*, E^*)$ as output. Algorithm 1 can be divided into 3 steps. Step1(line $1-2$) establishes the min-heap of N according to the execution order of statements. Step2(line $3-9$) folds those unrelating to memory management paths. Step3(line $10-12$) adjusts the $Red\_Black\_Tree(N^*)$ and the $Map(N^*)$, and obtains the final projection graph G^*). In this algorithm, $n.ind$ and $n.outd$ stand for the in-degree and out-degree of node n respectively. a means the node a is in set A. f means the node f is in F. Lastly, whenever a loop is added to G^* (even when the loop already exists), both the in-degree and the out-degree of the looping node are increased by one.

Considering the complexity of the projection algorithm, the Red Black Tree is used to store the nodes in set N. Red Black Tree is a data structure, which is an approximate balanced tree with only two types of nodes: red nodes and black nodes. This type of data structure has the advantage of high search efficiency, due to that it is sequential which can avoid the disorder during the search process.

In the algorithm, the set E in graph G (the execution sequence of statements in procedure P) is implemented as a set of Maps. A Map is a data structure stored in the form of key-value pairs. The elements in a Map are the ordering relation between a node n in set N and its successor nodes (i.e., nodes in $\Gamma(n)$).

The projection algorithm is acceptable in the aspect of complexity. In this algorithm, this paper assumes that the number of edges is linear in N, all the time complexity of Step1, Step2 and Step3 are $O(log(N))$, and the space complexity

Algorithm 1 ControlFlowProjection (G)

Input: A control flow graph G for program P.
Output: Projection graph G^* for the input control flow graph

```
1.  Initialize the Red Black Tree RBT(N).
2.  For all n from N, add n and Γ(n) to Map(n).
3.  while RBT has next element do
4.    if n.ind ≥ 2||n.outd ≥ 2 then
5.      if a ∈ Γ(n)||f ∈ Γ(n) then
6.        Γ(n) ← a||Γ(n) ← f.
7.    else then
8.      Γ(n) ← (n ∪ Γ(n) ∩ N*).
9.  end while
10. Delete all n that are not in N*.
11. RBT(N) ← RBT(N*).
12. Map(N) ← Map(N*).
```

of the three steps are $S(N)$($S(N)$ is linear). Therefore, the time complexity of this algorithm is $O(log(N))$ and the space complexity of this algorithm is $S(N)$.

B. Detection Method

Given a graph $G^* = (N^*, E^*)$, this paper defines the direct predecessor of a node as follows:

Definition 3. *Direct Predecessor Node For two nodes m, n ($m, n \in N^*$), if m is the first open node among all the predecessor nodes of node n (i.e., (n)), then m is called the direct predecessor node of n.*

The detection algorithm (Algorithm 2) takes the output of the projection process as input. Given the nodes in set A or F, the algorithm finds the direct predecessor node of all nodes by traversing the Red Black Tree. In this algorithm, the node $base$ is the last node in set A or F traversed to, i.e., $base$ is the currently visiting node. In line $7-12$, the first node from A or F will be pushed into the stack when the stack is empty. Otherwise it traverse the next node. In line $14-20$, there may be memory leaks when the direct predecessor node of $base$ is the node from B, according to the second case from Fig. 1 in Section II. Another situation is that the direct predecessor node of base is not the node from B, if the stack is not empty after the detection process, then there may be some memory spaces that have not been freed which may lead to memory leaks.

Take projection graph in Fig. 4 as an example. According to the detection algorithm, the direct predecessor node of node S_9 is node S_7, and the direct predecessor node of node S_7 is node S_6, node S_6 is a branch node. That is, the memory blocks are freed in the conditional branch of node 6, which is determined to be a suspected memory leak.

There are only one loop in the traversal of Red Black Tree, so the time complexity of this algorithm is $O(log(N))$, and the space complexity of this algorithm is $S(N)$.

Algorithm 2 MemoryLeakDetection (G^*)

Input: The projection graph G^*.
Output: The detection results for memory leaks ($true$ or $false$)

```
1.  Declare a stack to be detected.
2.  Declare node base and node current for traverse.
3.  Check_MemLeak(Tree RBT, Set Map)
4.  while RBT has next element do
5.    set base = null.
6.    set current = the current element of RBT.
7.    if base == null then
8.      if current != A && current != F then
9.        continue.
10.     else then
11.       base ← current.
12.       stack.push(base).
13.   else then
14.     if current ∈ B then
15.       if B.ind < 2||B.outd < 2 then
16.         return false.
17.       else if current == base then
18.         stack.push(current).
19.       else if current != base then
20.         stack.pop().
21. end while
22. if stack != null then
23.   return false.
24. else then
25.   return true.
```

C. Optimization Strategy

The control flow graph projection makes the detection process simple and intuitive. More importantly, all the cases of memory leaks mentioned in Section II can be detected in the detection algorithm. However, this approach may result in a high false positive rate, since it does not analyze the specific value of conditional predicate in the valuating of control flow branches, in short, it has a low sensitivity to data flows. In Fig. 5, the statement 5 will be marked with a B node (control flow branching statement). Apart from this, the example code will be judged to have a suspected memory leak by using the algorithm above, since the buffer may not be freed in every branch. However, there are no memory leaks in this example. Specifically, the result of the condition is true when analyzing the conditional predicate of line 5 (buffer != NULL), that is, the memory blocks are freed after being allocated.

In order to reduce such false positives, this paper combines symbolic execution[3] and constraint solving[4] during the execution of the algorithm to accurately evaluate the branching conditions.

Symbolic execution simulates the execution of programs

[3]Symbolic execution.https://en.wikipedia.org/wiki/Symbolic_execution.2017.
[4]Constraint solving. http://www.constraintsolving.com/. 2017.

```
0 void main(){
1  long *buffer;
2  size_t size;
3  buffer= (long*)malloc(1000*sizeof(long));
4  size = _msize(buffer);
5  if(buffer != NULL)
6  free(buffer);
7 }
```

Figure 5. A piece of code

by symbolizing variables. Specifically, symbolic execution collects memory allocation and release points by backward traversal for the CFG. At the same time, at each of the control flow branch, conditions of the branch are added to the current path conditions. Finally, those paths related to memory allocation and release will be merged at the meeting node of the CFG, and then according to Hoare logic[5], our system carries constraint solving to the merged paths. Constraint solving is accompanied by the projection process.

D. System Workflow

The input data of PML_Checker is C language source code. After lexical analysis and syntax analysis, abstract syntax tree generates the original control flow graph, which will be output to the system interface. Next, data flow analysis and symbolic execution extend the original CFG, constraint solving to the data flow conditions will find the dependencies from the memory allocation to deallocation, and the extended CFG will be output as a mid product. Then the projected CFG wil be produced at the base of extended CFG by projection algorithm. At last, according to the detection method mentioned at the approach section, PML_Checker detects memory leaks from the projected CFG, and displays the test results.

IV. Experimental Setup

To evaluate the projection approach proposed in Section III, this paper compares our tool—PML_Checker with three open-source tools which represent the existing different static detection approaches.

A. Test Tools

In our experiment, there are three state-of-the-art detection tools compared with PML_Checker: CppCheck[6], Splint[7] and RL_Detector. They are all open-source static detection tools, and have a good performance in detecting memory leaks.

Splint develops different detecting standards for different types of errors in the compilation process, it detects memory leaks by annotations to record the pointer objects lifetime [8]. CppCheck first creates symbol database of variables, functions and so on, and then it detects memory leaks by the embedded inspection classes. RL_Detector is a static detection tool implemented, which adopts resources streamlined slices construction and is achieved a comparative accurate analysis of memory leaks. This section compares our tool PML_Checker which implements the presented approach with the above tools.

B. Test Subjects

TABLE I
Information of real world repositories

Statistical metrics	SPEC CPU 2000	SIR
The number of test program	15	15
Total number of lines(kloc)	581	267.9
The average number of lines(kloc)	38.7	17.9
The line number of biggest program(kloc)	205.8	122.2

Table I lists the information of real world repositories. This experiment adopts two real world repositories for evaluation: SPEC CPU 2000[8] and SIR[9] (Software-artifact Infrastructure Repository). Specifically, we take 15 programs with 581 thousand lines code from SPEC CPU 2000, and 15 programs with 267.9 thousand lines code from SIR. However, we can not count the false negatives of the two repositories, because of the large amount of code.

TABLE II
Information of artificial cases

Statistical metrics	CC code		SARD cases	
	control flow	data type	bad	good
The number of test program	10	10	20	20
Total number of lines(Loc)	148	326	793	821
The average number of lines(Loc)	14.8	32.6	39.7	41.2
The line number of biggest program(Loc)	18	95	75	77

In order to count false negatives of PML_Checker, this experiment also uses artificial small code as test cases: SARD[10] (Software Assurance Reference Dataset) and CC code[11]. SARD is a public benchmark providing test cases. This experiment take 40 memory-related test cases, including 20 bad cases with memory errors and 20 corresponding good cases with no errors. CC code is provided by this paper to verify PML_Checker from two aspects: complex control flows and complex data types. A complex control flow refers to a control flow graph with more than one control flow branches, which is the focus of this paper. To make the experiment more convincing, this paper considers the complexity of data types. In our experiment, complex data types include data structure like linked list, struct, array and the combination of these data types. CC code presents 10 small programs with different

[5]Hoare logic. https://en.wikipedia.org/wiki/Hoare_logic. 2017.
[6]CppCheck. trac.cppcheck.net/wiki. 2016.
[7]Splint. http://www.splint.org/. 2010.
[8]SPEC. http://www.spec.org/cpu/. 2007.
[9]SIR. http://sir.unl.edu/content/sir.php. 2017.
[10]NIST. https://samate.nist.gov/SARD/. 2016.
[11]PML_Checker.https://github.com/sunxiaohuiczcz/PML_Checker.git.2017.

complex control flows, and 10 small programs with different complex data types. Adhering to the principle of single case coverage scenarios minimized[12], each case only covers one test scenario and includes only one memory leak. The detailed information of study cases are shown in Table II.

C. Metrics

There are seven metrics in our experiments for evaluation.

- *TW:* The total number of the reported memory leaks.
- *TL:* The total number of memory leaks contained in a program, and it takes the number of pointers pointing to memory blocks as the measurement standard.
- *NF:* The number of false positives.
- *MLF:* The difference of *TW* and *NF*, which denotes the number of real memory leaks in the test results.
- *FP:* The false positive rate, a metric used to measure the accuracy. It can be calculated by the following formula: $FP=\frac{NF}{TW}$.
- *NP:* The false negative rate, a metric used to measure the accuracy. It can be calculated by the following formula: $NP=1-\frac{TW}{TL}$.
- *RunTime:* The run time of each tool for C program.

V. Evaluation

In general, this paper investigates the following research questions to evaluate PML_Checker:

RQ1: How does our approach perform under the real world repositories and public benchmarks?

RQ2: How does our approach perform in terms of effectiveness analysis for complex control flows?

RQ3: How does our approach perform in terms of effectiveness analysis for complex data types?

A. RQ1: Performance on real world repositories and public benchmarks

1) Accuracy Analysis:

This section includes two-part experiments: experiment on large amount code (SPEC CPU 2000 and SIR), experiment on small amount code (test cases from SARD).

Table III shows the test results on SPEC CPU 2000 and Table IV shows the test results on SIR. Compare PML_Checker with other three tools, we have the following observations.

First, calculate the *FP* in Table III. *FP*(CppCheck) $\approx$ 11%, *FP*(Splint) = 44%, *FP*(RL_Detector) = 9%, *FP*(PML_Checker) = 13%. Similarly, in Table V, *FP*(CppCheck) = 19%, *FP*(Splint) = 0.30%, *FP*(RL_Detector) = 8%, *FP*(PML_Checker) = 18%. In the view of *TW* from Table III and Table IV, Splint reports most bugs, and PML_Checker ranks second, while CppCheck reports the least bugs. However, in the view of *FP*, the *FP* of Splint is highest, and the *FP* of the other three tools are acceptable.

Second, this paper analyzes the effectiveness of the PML_Checker by comparing the MLF of test results. Test results from Table III can be converted into MLF of test results by constructing a matrix. Specifically, the results from Table IV can be abstracted to a sparse matrix consists of 15 quaternions. This paper compresses this matrix by removing all the rows that test results are zero, and a row in the matrix corresponds to a row in the table of test results. The final matrix D_1 is as follows.

TABLE III
Test results on SPEC CPU 2000

Program	Size (Kloc)	CppCheck		Splint		RLDetector		PMLChecker	
		TW	NF	TW	NF	TW	NF	TW	NF
gzip	7.8	1	1	1	1	1	1	3	1
vpr	17.0	0	0	0	0	0	0	4	1
gcc	205.8	1	0	46	24	35	0	22	1
mesa	49.7	1	0	9	5	4	2	17	5
art	1.3	1	0	0	0	1	0	3	0
mcf	1.9	0	0	5	2	0	0	1	1
equake	1.5	0	0	0	0	0	0	8	0
crafty	18.9	0	0	0	0	0	0	0	0
ammp	13.3	12	0	22	4	20	0	22	2
parser	10.9	0	0	0	0	0	0	0	0
perlbmk	58.2	2	1	0	0	4	1	2	0
gap	59.5	0	0	0	0	0	0	0	0
vortex	52.7	0	0	0	0	0	0	0	0
bzip2	4.6	1	0	2	1	0	0	1	0
twolf	19.7	0	0	0	0	0	0	0	0
Total	581	19	2	85	37	65	6	83	11

TABLE IV
Test results on SIR

Program	Size (Kloc)	CppCheck		Splint		RLDetector		PMLChecker	
		TW	NF	TW	NF	TW	NF	TW	NF
bash	59.8	7	1	15	3	3	0	17	2
flex	10.5	1	0	2	1	2	1	2	1
grep	10.1	3	1	0	0	0	0	2	0
gzip	5.7	0	0	0	0	0	0	0	0
make	35.5	1	0	0	0	0	0	1	0
print	0.7	0	0	0	0	0	0	0	0
print2	0.6	0	0	0	0	0	0	0	0
replace	0.6	0	0	0	0	0	0	0	0
schedule	0.4	0	0	5	2	0	0	2	0
schedule2	0.4	0	0	3	0	0	0	2	1
sed	14.4	4	2	0	0	0	0	3	0
space	6.2	2	0	0	0	0	0	1	1
tcas	0.2	0	0	0	0	0	0	0	0
totinfo	0.6	0	0	0	0	0	0	0	0
vim	122.2	3	0	12	5	8	0	4	1
Total	267.9	21	4	37	11	13	1	34	6

$$D_1 = \begin{bmatrix} 0 & 0 & 35 & 2 & 1 & 0 & 0 & 20 & 3 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 12 & 1 & 1 \\ 0 & 0 & 22 & 4 & 0 & 3 & 0 & 18 & 0 & 1 \\ 2 & 3 & 21 & 12 & 3 & 0 & 8 & 20 & 2 & 1 \end{bmatrix}^T$$

Similarly, D_2 is the matrix abstracted from Table IV, and D_2 is shown as follow:

$$D_2 = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 6 & 1 & 2 & 1 & 0 & 0 & 2 & 2 & 3 \\ 12 & 1 & 0 & 0 & 3 & 3 & 0 & 0 & 7 \\ 15 & 1 & 2 & 1 & 2 & 1 & 3 & 0 & 3 \end{bmatrix}^T$$

In Fig. 6, the abscissa shows the name of the program corresponding to each row in the matrix, and the ordinate shows the value of corresponding MLF. This figure displays the number of memory leaks confirmed by manually checking,

[12]Test Case Design. http://ecomputernotes.com/software-engineering/test-case-design. 2017.

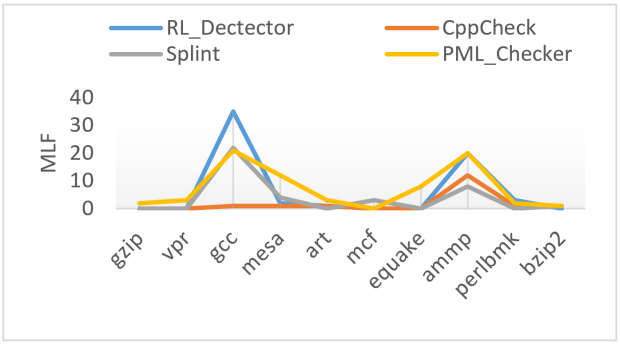

Figure 6. The MLF of each tool on SPEC CPU 2000

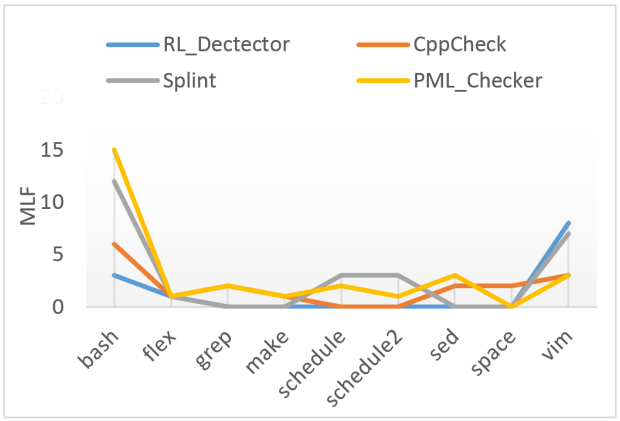

Figure 7. The MLF of each tool on SIR

among the test results on the 15 programs from SPEC CPU 2000. We conclude from Fig. 6 that PML_Checker has a good detection result on the test sets relatively. There is a big difference from test results for *gcc*. *gcc* is larger than others in size. Therefore, we deduce that the scalability of the PML_Checker and CppCheck for large scale test objects needs to be improved.

In Fig. 7, due to the smaller amount code of SIR comparing to SPEC CPU 2000 and fewer *TW* of each tool, the advantage of PML_Checker is not obvious. But the analysis results between the two benchmarks are generally consistent. By testing on SPEC CPU 2000 and SIR, PML_Checker shows a good performance in testing memory leak.

Table V shows the test results on 40 test cases from SARD. Compare PML_Checker with other three Tools, we have the following observations. First, calculate the *NP* of the test results of 20 test cases with memory errors marked "bad", and the *FP* of the test results of 20 test cases without memory errors which corresponding to the "bad" cases marked "good". *NP*(RL_Detector) = 70%, *FP*(RL_Detector) = 0. *NP*(CppCheck) = 50%, *FP*(CppCheck) = 0. *NP*(Splint) = 80%, *FP*(Splint) = 5%. *NP*(PML_Checker) = 30%, *FP*(PML_Checker) = 0. The *FP* of PML_Checker is the lowest among the four tools. For the analysis of *FP*, there are no false positives in the results of RL_Detector, CppCheck and PML_Checker. While Splint reports a false positive. This experiment fails to compare the *FP* due to the small size of program, but the test results on *NP* reflects the accuracy of PML_Checker in detecting memory leaks.

TABLE V
TEST RESULTS ON SARD TEST CASES

Cases	CppCheck (TW)	Splint (TW)	RLDetector (TW)	PMLChecker (TW)
bad	10	4	6	14
good	0	1	0	0

2) Efficiency Analysis:

Considering CppCheck, RL_Detector and PML_Checker all use symbolic execution, it is necessary to verify whether our approach affects the efficiency. This experiment chooses SPEC CPU 2000 as the test object due to its large amount of code and large number of files, and analyzes the run-time of the three tools.

Table VI lists the run-time of CppCheck, RL_Detector and PML_Checker. By comparing the run-time of these three tools, we conclude that the performance of RL_Detector and PML_Checker is better than CppCheck in run-time, and our approach does not affect the efficiency of PML_Checker.

TABLE VI
RUN TIME ON SPEC CPU 2000

Program	Size (Kloc)	RunTime(s)		
		CppCheck	RLDetector	PMLChecker
gzip	7.8	5.3	4.5	3.2
vpr	17.0	10.0	7.5	7.6
gcc	205.8	167.7	48.3	41.0
mesa	49.7	51.0	33.1	34.8
art	1.3	0.3	0.9	0.4
mcf	1.9	1.0	4.0	3.7
equake	1.5	0.2	1.0	0.4
crafty	18.9	27.3	14.2	13.3
ammp	13.3	7.4	10.3	10.0
parser	10.9	4.0	6.7	5.1
perlbmk	58.2	123.2	41.3	39.0
gap	59.5	29.0	20.9	20.0
vortex	52.7	73.2	30.1	26.7
bzip2	4.6	2.0	0.6	1.5
twolf	19.7	11.0	23.7	24.0
Total	581.0	512.6	247.1	230.7

B. RQ2: Effectiveness Analysis of Complex Control Flows

TABLE VII
TEST RESULTS ON COMPLEX CONTROL FLOWS

Cases	CppCheck	Splint	RLDetector	PMLChecker
branch1	√	×	√	√
branch2	√	√	√	√
loop1	×	×	√	√
loop2	√	×	√	√
chainBranch1	×	×	√	√
chainBranch2	×	×	√	√
chainLoop1	√	√	×	√
chainLoop2	√	√	×	√
nestingBranch	×	×	√	√
nestingLoop	×	×	√	√

Table VII lists the test results on 10 small programs. Comparing PML_Checker to other threetools, we have

the following observations. First, calculate the *NP* of the test results. *NP*(CppCheck) = 50%, *NP*(Splint) = 70%, *NP*(RL_Detector) = 20%, *NP*(PML_Checker) = 0. PML_Checker covered all the test cases about complex control flows. In other words, the results reflect the effectiveness of PML_Checker on complex control flows. Second, regarding the comparison of the approaches. Obviously, our approach has advantage of complex-control-flow detection. The regular match based approach matches the source code with the vulnerabilities in the process of detection. This approach is not sensitive and inflexible to complex control flow. The style and notation based approach improves the software quality by improving the programming style, and discoves the potential bugs of program. However, this kind of analysis is not complete, so it has a high false negative rate. The approach by constructing resources streamlined slices only makes a simple assumption for allocation and deallocation of resources within loop bodies. It reduces the number of loops to 1 and treats the loops the same as branches. Therefore, this approach can not pass the test cases on the chain_loop structure (chainLoop1 and chainLoop2) in the experiment.

C. RQ2: Effectiveness Analysis of Complex Data Types

TABLE VIII
TEST RESULTS ON COMPLEX DATA TYPES

Cases	CppCheck	Splint	RLDetector	PMLChecker
array1	√	×	√	√
array2	×	×	×	×
array3	×	×	×	√
list1	×	×	×	×
list2	×	√	×	√
list3	×	√	×	√
struct1	×	√	×	√
struct2	×	√	×	√
arrayStruct1	×	√	√	√
arrsyStruct2	√	√	√	√

Table VIII lists the test results on 10 small programs. Comparing our approach with other approaches, we have the following observations. First, calculate the *NP* of the test results. *NP*(CppCheck) = 80%, *NP*(Splint) = 40%, *NP*(RL_Detector) = 70%, *NP*(PML_Checker) = 20%. The results of Splint and PML_Checker show a higher effectiveness. Splint has a high effectiveness because it mainly checks the specifications in programs, since it is sensitive to different data types. PML_Checker shows a high effectiveness, because it simplifies the analysis for complex data types in the process of abstracting the control flows. Second, comparing PML_Checker to Splint, Splint reports memory leaks on the latter three data types (linked list, struct and array_struct), except the memory leaks in an array. Due to the programs that are provided in this paper are in small scale and simple data relationships, PML_Checker shows a little advantage compared with Splint. For the former approaches, that is CppCheck and RL_Detector, there is a large space to improve in analyzing the complex data types.

D. Summary

From the above experimental results, we have the following main findings including advantages and limitations:

- For the accuracy of detection, PMl_Checker shows a lower false negative comparing to other three tools, while it shows a high false positive rate in real world repositories (SPEC CPU 2000 and SIR). Therefore, in the next step, more detailed data flow analysis are plan to be added into our approach.
- For the complex control flows and complex data types analysis, PML_Checker is better than the other three tools.
- For the scalability of detection, our experiment needs to expand the range of detection. Specifically, it is possible to detect the large scale benchmarks with millions of code lines, or source code from some open-source software.

VI. RELATED WORK

This section mainly discusses three aspects of research: Research on static approaches to memory leaks, research on complex programs analysis and research on the application of projection method.

A. Static Approaches on Memory Leaks

In recent years, some researchers proposed new static approaches to memory leak detection. [9], [10] and [11] detect errors by model checking. [9], [10] proposed Memory State Transition Graph (MSTG) and implemented the tool—Melton. While [11] considered from the perspective of object ownership and modeling it. But these approaches may cause high positive rate in some cases. [12] presented heap memory abstract method for heap-manipulating programs. For complex programs, our approach show higher accuracy at detecting memory leaks.

We also survey the static detection tools particularly for memory leaks detection. Noted that one tool may adopt multiple approaches. We highlight some well-known tools as follows. One of them is Prefix [13], which is embedded in the Visual Studio. Prefix detects memory leaks by symbolic simulation. It is path-sensitive and uses function summaries for scalability. However, Prefix explores one path at a time, which becomes inefficient when procedures have many paths. Some other tools show a high analysis accuracy by using different techniques. For instance, Splint and CppCheck use the rule-based approach, where Splint mainly analyzes style and annotations of program, and CppCheck detects errors by matching expressions. As an example of flow-sensitive approach, tool Cqual[13] uses type inference and constraint solving techniques for improve accuracy. These tools essentially check the potential memory leaks by adding annotations for the access operation in the source programs. This approach has high preciseness, however, it will generate many false positives as the complexity of the program increases, because the annotations added cannot be reused. Tool RL_Detector [14]

[13]Cqual. http://www.cs.umd.edu/ jfoster/cqual/. 2004.

detects resource leaks in C programs. It improves detection efficiency by constructing resource behavior to streamline slices, this method retains all statements that are allocated and deallocated of resources, as well as the statements that affect the use of the resource (including memory). Our tool is based on [5], and combines approaches such as symbolic execution, which are used in the above tools as well. Compared with existing work, our tool is specific for detecting memory leaks in C programs, especially in the program with complex control flows.

B. Complex Programs Analysis

Research on the analysis complex programs mainly focus on control flow graphs. To our knowledge, there is currently no investigation of memory leak detection in complex control flow. [2] and [15] study control flow graphs. The DMP (diverge-merge processor) in [2] is a processor architecture to predict complex branches dynamically. [15] presents a method to calculate all the worst paths from any node. There are other studies on complex object-oriented programs like [16], [17]. Specifically, the research object of [16] are complex path conditions, and the research object of [17] are inheritance and friend functions in object-oriented languages. [18] shows a method to measure the complexity of programs. This paper combines complex control flow with memory leaks in the program, shows a novel detecting method.

C. The Application of Projection Method

Recently, the idea of projection is often applied to computer vision [19], [20], mathematical model selection [21] and other related research fields due to its close relationship with graphs. There are only a few of papers use the projection method in the program analysis. [22] analyzes the specification and abortion of programs by the projection of functions. While in terms of safety, this paper only takes a conservative analysis, in other words, the output is "uncertain or "unknown for some uncertain input, which is relatively fuzzy, so the accuracy of the analysis results need to be improved. In our approach, we redefine the projection of the program control flow graphs, and solve safety problems with a specific algorithm for detecting memory leaks.

VII. Conclusion

In this paper, we focus on memory leak detection with complex control flows. We propose a projection-based approach to increase the accuracy in detecting memory leaks in C source code. We implement a detection tool named PML_Checker, and evaluate our approach by comparing with three open-source tools (CppCheck, Splint, RL_Detector) on real world repositories (SPEC CPU 2000, SIR), public benchmarks (SARD) and CC code. Experimental results show that our approach has lower false negatives than three other approaches, especially in C source code with complex control flows and complex data types. The future direction is to improve our approach in terms of scalability. Specifically, it is possible to improve the detection accuracy by combining with other static methods such as data flow analysis.

References

[1] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *Proc. 30th Annual International Computer Software and Applications Conference - COMPSAC*, 2006, pp. 343–350.

[2] H. Kim, J. A. Joao, and O. Mutlu, "Diverge-merge processor (DMP): dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO*, 2006, pp. 53–64.

[3] M. Li, Y. Chen, L. Wang, and G. H. Xu, "Dynamically validating static memory leak warnings," in *Proc. International Symposium on Software Testing and Analysis, ISSTA*, 2013, pp. 112–122.

[4] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proc. International Symposium on Software Testing and Analysis - ISSTA*, 2012, pp. 254–264.

[5] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 115–125.

[6] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, no. 2, pp. 139–156, 2001.

[7] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *Proc. 13th International Symposium on Static Analysis - SAS*, 2006, pp. 405–424.

[8] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42–51, 2002.

[9] Z. Xu, J. Zhang, and Z. Xu, "Memory leak detection based on memory state transition graph," in *Proc. Asia-Pacific Software Engineering Conference - APSEC*, 2011, pp. 33–40.

[10] ——, "Melton: a practical and precise memory leak detection tool for c programs," *Frontiers of Computer Science*, no. 1, pp. 34–54, 2015.

[11] D. L. Heine and M. S. Lam, "Static detection of leaks in polymorphic containers," in *Proc. International Conference on Software Engineering - ICSE*, 2006, pp. 252–261.

[12] L. Dong, J. Wang, and L. Chen, "Modular heap abstraction-based memory leak detection for heap-manipulating programs," in *Proc. 19th Asia-Pacific Software Engineering Conference - APSEC*, 2012, pp. 20–29.

[13] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software - Practice and Experience - SPE*, vol. 30, no. 7, pp. 775–802, 2000.

[14] X.-J. Ji, "Research on the static analysis of resource leak fault," Ph.D. dissertation, College of Computer and Control Engineering, Nankai University, Tianjin, China, 2014, http://cdmd.cnki.com.cn/Article/CDMD-10055-1015528928.htm, accessed at 25 June 2017.

[15] J. C. Kleinsorge, H. Falk, and P. Marwedel, "Simple analysis of partial worst-case execution paths on general control flow graphs," in *Proc. ACM International Conference on Embedded Software - EMSOFT*.

[16] X. Li, Y. Liang, and H. Q. etc., "Symbolic execution of complex program driven by machine learning based constraint solving," in *Proc. Conference on Automated Software Engineering - ASE*, 2016, pp. 554–559.

[17] J. Mohapatra, S. Gid, K. Debasis, and S. K. Das, "Slicing complex c++ program dynamically," in *Proc. International Conference on Advanced Communication Control and Computing Teclmologies - ICACCCT*, 2014, pp. 1765–1770.

[18] B. Katzmarski and R. Koschke, "Program complexity metrics and programmer opinions," in *Proc. 20th International Conference on Program Comprehension - ICPC*, 2012, pp. 17–26.

[19] D. Dai and L. V. Gool, "Ensemble projection for semi-supervised image classification," in *Proc. International Conference on Computer Vision - ICCV*, 2014, pp. 2072–2079.

[20] Z. Huang, R. Wang, and S. Shan, "Projection metric learning on grassmann manifold with application to video based face recognition," in *Proc. Conference on Computer Vision and Pattern Recognition - CVPR*, 2015, pp. 140–149.

[21] D. J. Nott and C. Leng, "Bayesian projection approaches to variable selection in generalized linear models," *Computational Statistics & Data Analysis*, vol. 54, no. 12, pp. 3227–3241, 2010.

[22] K. Davis, "Projection-based program analysis," Ph.D. dissertation, Department of Computing. University of Glasgow, Glasgow, Scotland, UK, 1994.