A Projection-based Approach for Memory Leak Detection

Xiaohui Sun[*], Chenkai Guo[†], Naipeng Dong[‡], Quanqi Ye[‖], Jing Xu[§], Xiujuan Ji[¶]

[¶]Department of Binhai
Nankai University
[¶]jixiujuan@mail.nankai.edu.cn

[*†§]Department of Computer and Control Engineering
Nankai University
[*]2120150397@mail.nankai.edu.cn
[†]guochenkai88@gmail.com
[§]xujing@nankai.edu.cn

[‡‖]School of Computing
National University of Singapore
[‡]dcsdn@nus.edu.sg
[‖]yequanqi@u.nus.edu

***Abstract*—One of the major causes of software safety issues is memory leak; and the safety issues caused by memory leak are highly hazardous. Therefore, in order to ensure the software quality, memory leak detection is necessary so that it can be corrected. However, detecting memory leak vulnerabilities is challenging in program analysis, as they are well concealed. Particularly, existing detection tools are weak in efficiency and accuracy especially when the targeted program contains multiple branches with complex control flows. In this paper, we propose a projection-based approach to detect memory leaks in the source code of programs with multiple control flow branches. This approach projects the original control flow graph to a simpler one according to the features of memory allocation and deallocation in C/C++ source code, which reduces the analysis complexity.**

We implemented a memory leak detection tool PML_Checker and evaluated the tool by comparing with three other static detection tools like CppCheck on both public benchmarks and some specific test cases. The detection results show that PML_Checker reported the most memory leak vulnerabilities among the other tools in source code of programs with complex control flow and complex data types, and PML_Checker obtained higher efficiency and accuracy on public benchmarks.

I. Introduction

Memory leak is well-known to be a major cause of reliability and performance issues in software, for example in embedded systems, as it causes the long-running applications or embedded systems eventually 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, in September 1992 in London, a memory leak in the ambulance service system [?] resulted in a system crash, which led to the system breakdown after three weeks of continuous program operating. On October 22, 2012, a potential memory leak caused Amazon's EC2 cloud services to be partially disrupted, which affected the operations of hundreds of EC2 customers[1]. 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, as the only symptom of memory leak is the slow increasing in memory consumption.

[1]http://iguowei.com/2012/11/22/amazon-service-event/. 2017.

To address this challenge, there are two general techniques in existing works - dynamic testing and static source code analysis [?]. Compared to dynamic testing analysis which relies on the coverage of the test cases and requires long-time execution, 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, we focus on detecting memory leaks by applying source code analysis. Among program source codes written in various languages, detecting memory leak is particularly necessary when the program is written in low-level programming languages such as C/C++. Because the low-level programming languages allow manual memory management, for example explicit memory allocation and deallocation [?]. Such manual memory operations are often error-prone which lead to vulnerabilities.

The root cause of vulnerabilities can be summarised as the relation between memory allocation and deallocation directly or indirectly caused by explicit memory operations, that is whether the following two points have been followed [?].

- The dynamic memory blocks are not free/deallocated.
- The dynamic memory blocks are freed in wrong order.

To check the above two errors, there are in general the following 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 via assignments for all memory locations represented by both top-level and address-taken pointers, which is also known as VFG (Value Flow Graph). While the basic idea of control-flow-based approach is to construct a model that represents the dynamic memory allocation and deallocation on the CFG (Control Flow Graph). Based on the CFG model, we check whether a block of heap memory space is reclaimed by the program or the runtime system when the lifetime of the program has ended - if a block is not reclaimed, then there is memory leak. Compared the VFG model [?] 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, we take into account the memory pointers' lifecycle, i.e., including all the memory operations, in order to capture the indirect allocation-deallocation relation and thus obtain accurate results.

The existing control-flow-based approach has 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 flows 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, we focus on efficiently and accurately detecting memory leaks in programs with complex control flows.

The higher complexity of the control flows is, the easier the vulnerabilities can hide. To detect memory leaks, we need 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 a programs control flows can greatly improve the efficiency and accuracy of detecting memory leaks in the program. Therefore, in this work, we propose to apply projection to the control flow graph of a program. The projection abstracts the original control flow graph to a simplified graph so 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 a program has multiple control flow branches, we need 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. To do so, we adopt path-sensitive analysis method [?], which records the different program states of the two branch paths by tracing each branch of the program control flow. This method analyzes the combinatorial relation between branches, which can be used to 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, we use a tool named PAT (Path Analysis and Testing) [?], which uses symbolic execution and constraint solving, to perform accurate evaluation on branch evaluations. As a result, we propose a set of rules for guiding the projection which essentially reduces irrelevant nodes in a control flow graph, and we prove the correctness of the rules - no irrelevant path is considered and no relevant path is ignored.

In the next section we introduce our control flow graphs projection approach and prove its effectiveness. In Section II, we analyze 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, we conduct some experiments to test PML_Checker by comparing with other detection tools, and we evaluate it at its effectiveness, accuracy, run time and scalability. Section V discusses related research. And finally we conclude this work in Section VI.

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 in existing works [?]:

- *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.

In this paper, we focus on the second view - reachability, as reasoning on the reachability is more intuitive in the CFG, which our analysis is based on. 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 a control flow graph analysis on the correspondence relation between the memory pointers and the lifetime of the corresponding memory spaces, referred to as the reachability of the pointers.

As stated in Section I, when procedure P has multiple 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, this increases the complexity of the control flow graph analysis, because of the complex branch guarding condition analysis. We formally define 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 flow. 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, we need 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 space 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, which 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.

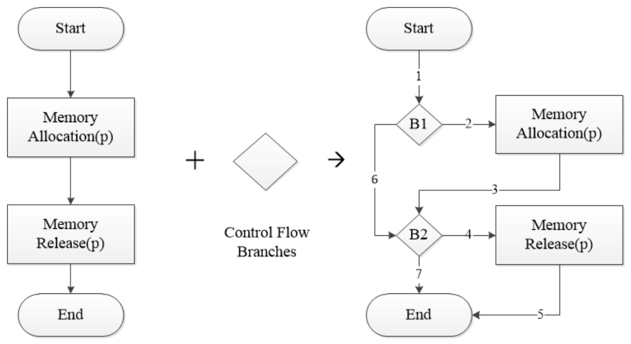

Fig. 1. Description of memory leaks reasons

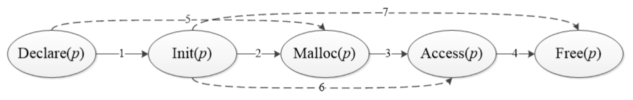

Fig. 2. Lifetime of a pointer p

In each branch of the control flow graph, in addition to considering the allocation and deallocation of memory spaces, we need 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 multiple pointers are considered, the cases that may lead to memory issues are complicated, which is another challenge in the memory leak detection.

III. Approach

Similar to the CFG based static analysis, our approach also first constructs a detection model of C program, where memory leak may happen. The main difference of our approach is that, we perform projection in constructing the detection model, which simplifies the later 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. First of all, we introduce 3 definitions which are going to be used in the projection rules.

Definition 2 (Projection Subject). *Projection subject is the control flow graph of the program to be analyzed, which is a directed graph $G = (N, E)$, 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.*

Definition 3 (Projection Target). *Projection target is a directed graph G^* 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 4 (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, we summarize 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.

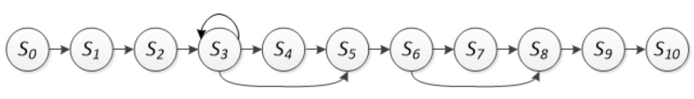

Fig. 3. The CFG of the program (G)

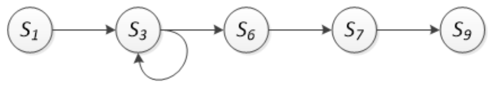

Fig. 4. The projection graph (G*)

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. In the algorithm, $n.ind$ and $n.outd$ stand for the in-degree and out-degree of node n respectively. $A.a$ means the node a is in set A. $F.f$ 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 Maps 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, we assume 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 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 $S(N)$.

Algorithm 1 ControlFlowProjection (G)

```
Input: A control flow graph $G$ for program $P$.
Output: Projection graph $G^*$ for the input control flow graph
Begin
Step1: (According to the execution order of statements, establish the min-heap of $N$)
1.  define $Red\_Black\_Tree(N)$;
2.  init $Red\_Black\_Tree(N)$;
3.  create $Red\_Black\_Tree(N)$;
4.  $\forall n \in N, Map(n) \leftarrow (n, \Gamma(n))$;
Step2: (Fold paths)
1.  while $Red\_Black\_Tree$.hasNext
2.  do begin
3.  if $n.ind \geq 2 || n.outd \geq 2$ then
4.      if $A.a \in \Gamma(n) || F.f \in \Gamma(n)$
5.      then $\Gamma(n) \leftarrow A.a || \Gamma(n) \leftarrow F.f$
6.  else
7.      then $\Gamma(n) \leftarrow (n \cup \Gamma(n) \cap N^*)$
8.  end
Step3: (Adjust the $Red\_Black\_Tree(N^*)$ and the $Map(N^*)$, and obtain the final projection graph $G^*$)
1.  delete $n (n \notin N^*)$;
2.  $Red\_Black\_Tree(N) \leftarrow Red\_Black\_Tree(N^*)$
3.  $Map(N) \leftarrow Map(N^*)$;
End
```

B. Detection Process

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

Definition 5. *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 lines $8-13$, the first node from A or F will be pushed into the stack when the stack is empty. Otherwise we traverse the next node. In lines $15-21$, 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.

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 $S(N)$.

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

Algorithm 2 MemoryLeakDetection (G^*)

```
Input: The projection graph $G^*$.
Output: The detection results for memory leaks ($true$ or $false$)
Begin
1.  define Stack $stack$;
2.  define Node $base$, $current$;
3.  bool $MemLeak\_Check$(Tree $Red\_Black\_Tree$, Set $Map$)
4.  while $Red\_Black\_Tree$.hasNext
5.  do begin
6.      $base \leftarrow$ null;
7.     $current = Red\_Black\_Tree.current\_Ele$
8.     if $base$ == null then
9.         if $current$ != $A$ && $current$ != $F$ then
10.            Continue
11.        else then
12.            $base \leftarrow current$;
13.            $stack$.push($base$);
14.    else then
15.        if $current \in B$ then
16.            if $B.ind < 2 || B.outd < 2$ then
17.                return $false$;
18.            else if $current == base$ then
19.                $stack$.push($current$);
20.            else if $current$ != $base$ then
21.                $stack$.pop();
22. end
23. if $stack$ != null then
24.    return $false$;
25. else then
26.    return $true$;
End
```

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.

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, which leads to 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[2] and constraint solving[3] during the execution of the algorithm to accurately evaluate the branching conditions.

```
void main(){
 long *buffer;
 size_t size;
 buffer= (long*)malloc(1000*sizeof(long));
 size = _msize(buffer);
 if(buffer != NULL)
 free(buffer);
}
```

Fig. 5. A piece of code

Symbolic execution simulates the execution of programs by symbolizing variables. Specifically, symbolic execution collects memory allocation and release points by backward traversal of 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. Constraint solving uses the constraint solver to solve the satisfiability of the path conditions. Constraint solving is carried on the projected CFG. Therefore, symbolic execution and constraint solving can help to reduce similar false positives like the problem in this example.

D. System Framework

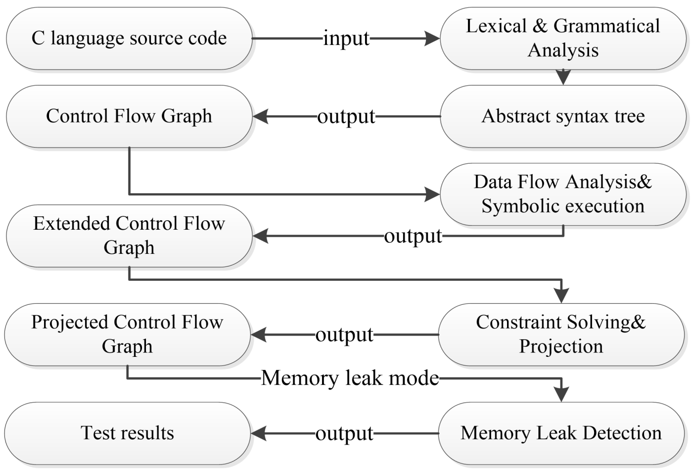
C language source code
input
Lexical & Grammatical Analysis
Control Flow Graph
output
Abstract syntax tree
Data Flow Analysis& Symbolic execution
Extended Control Flow Graph
output
Projected Control Flow Graph
output
Constraint Solving& Projection
Memory leak mode
Test results
output
Memory Leak Detection

Fig. 6. Framework flowchart for PML_Checker

Fig. 6 shows the flowchart for PML_Checker. The input data of PML_Checker is C language source code. after lexical analysis and syntax analysis, the CFG matched the source code is generated by the abstract syntax tree. Data flow analysis and symbolic execution extend the original CFG, the 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 shows the test results.

IV. Experiments

To evaluate the projection approach proposed in Section III, we compared our tool – PML_Checker with other tools which represent the existing different static detection approaches. For the sake of fairness, in the results of our experiments, we considered the effectiveness of the approach from two aspects: complex control flows and complex data types. In details, a CFG with more than one control flow branches of the procedure P is called a complex control flow. Data structure like linked list, struct, array and the combination of these data types are called complex data types. Furthermore, we adopt SPEC CPU 2000[4], SIR[5] (Software-artifact Infrastructure Repository) and some test cases about memory errors from the SARD[6] (Software Assurance Reference Dataset) for measuring the accuracy of the projection approach.

In general, we investigate the following research questions in our experiments:

RQ1: How does our approach perform compared with existing approaches in terms of effectiveness analysis for complex control flows?

RQ2: How does our approach perform compared with existing approaches in terms of effectiveness analysis for complex data types?

RQ3: How does our approach perform compared with existing approaches under the public benchmarks in terms of accuracy and run time analysis?

A. Setup

In the rest of this section, we present detailed information of the test subjects (Section IV-A1), selected approaches to compare with (Section IV-A2), the parameters and metrics (Section IV-A3) in our experiments.

1) Test Subjects: We evaluated our approach on a number of small programs with memory leaks that have different complex control flows and complex data types. In addition, a collection of larger programs is tested in our experiments, which include all the C programs from SPEC CPU 2000 15 in total, all the C programs from SIR 15 in total, and 40 test cases about memory errors from SARD.

2) Selected Approaches: Table I lists the 4 approaches considered in our experiments. There are several reasons for selecting the following approaches and the corresponding tools. Regular matching and style, notation based detection are common approaches adopt in source code static detection.

[2]Symbolic execution. https://en.wikipedia.org/wiki/Symbolic_execution. 2017.

[3]Constraint solving. http://www.constraintsolving.com/. 2017.

[4]SPEC. http://www.spec.org/cpu/. 2007.

[5]SIR. http://sir.unl.edu/content/sir.php. 2017.

[6]NIST. https://samate.nist.gov/SARD/. 2016.

TABLE I
INFORMATION OF APPROACHES

No.	Abbreviation	Description	Tool
A1	exp-mat	expression matching	CppCheck
A2	stl-not	style and notation	Splint
A3	steam-slic	resources streamlined slices	RL_Detector
A4	pro-grap	CFG projection	PML_Checker

Both CppCheck[7] and Splint[8] are open source static detection tools used in Windows operating system. Apart from this, both tools can detect memory leaks in C programming language. In details, 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 life-time [?]. 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. We compare our tool PML_Checker which implements the presented approach with the above tools.

3) Parameters and Metrics: There are some parameters and metrics used 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.
- *FP:* The number of false positives.
- *MLF:* The difference of *TW* and *FP*, which denotes the number of real memory leaks in the test results.
- *FPR:* The false positive rate, a metric used to measure the accuracy. It can be calculated by the following formula: $FPR=\frac{FP}{TW}$.
- *FNR:* The false negative rate, a metric used to measure the accuracy. It can be calculated by the following formula: $FNR=1-\frac{TW}{TL}$.
- *Time:* The run time of each tool for C program.

B. Experimental Results

RQ1: *Effectiveness for Complex Control Flow*

The first research question is to compare the effectiveness achieved by the 4 tools with respect to complex control flows. Table II lists the test results on 9 small programs and Fig. 7 shows the number of the 4 tools reported memory leaks in each control flow structure. Comparing our approach with other approaches, we have the following observations.

First, PML_Checker covered all the test cases about complex control flow in this experiment. In other words, the *FNR* of the CFG projection approach is lower than other approaches for detecting memory leaks in complex control flow, which reflects the effectiveness of this approach for complex control flow.

[7]CppCheck. trac.cppcheck.net/wiki. 2016.
[8]Splint. http://www.splint.org/. 2010.

TABLE II
TEST RESULTS ON COMPLEX CONTROL FLOWS

Test Case No.	Test Cases	A1	A2	A3	A4
C_case1	Simple_branch	YES	NO	YES	YES
C_case2	Simple_loop_1	NO	NO	YES	YES
C_case3	Simple_loop_2	YES	NO	YES	YES
C_case4	Chain_branch_1	NO	NO	YES	YES
C_case5	Chain_branch_2	NO	NO	YES	YES
C_case6	Chain_loop_1	YES	YES	NO	YES
C_case7	Chain_loop_2	YES	YES	NO	YES
C_case8	Nesting_branch	NO	NO	YES	YES
C_case9	Nesting_loop	NO	NO	YES	YES
FNR		**56%**	**78%**	**22%**	**0**

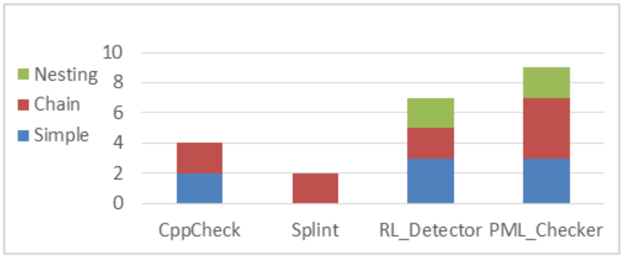

Fig. 7. Description of memory leaks reasons

Second, regarding the comparison of the approaches, the regular match based approach compares the source code with the vulnerabilities in the process of detecting memory leaks. This approach is not sensitive to complex control flow and inflexible in the types of vulnerabilities to compare with. The style and notation based approach improves the software quality by improving the programming style and discovering the potential bugs that impacting the portability of programs. However, the memory leak 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 (C_case6 and C_case7) in the experiment.

RQ2: *Effectiveness for Complex Data Flow*

The second research question is to compare the effectiveness achieved by the 4 approaches with respect to complex control flows. Table III lists the test results on 10 small programs and Fig. 8 shows the comparison on the number of memory leaks in each data type by the 4 tools. Comparing our approach with other approaches, we have the following observations.

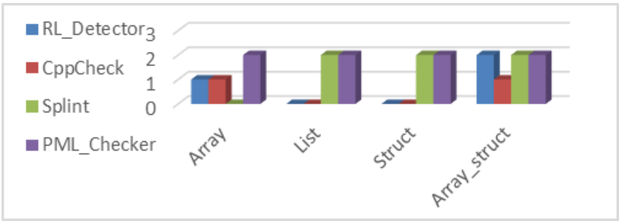

Fig. 8. The leaks number of each data type

TABLE III
TEST RESULTS ON COMPLEX DATA TYPES

Test Case No.	Test Cases	A1	A2	A3	A4
T_case1	Array_1	YES	NO	YES	YES
T_case2	Array_2	NO	NO	NO	NO
T_case3	Array_3	NO	NO	NO	YES
T_case4	List_1	NO	NO	NO	NO
T_case5	List_2	NO	YES	NO	YES
T_case6	List_3	NO	YES	NO	YES
T_case7	Struct_1	NO	YES	NO	YES
T_case8	Struct_2	NO	YES	NO	YES
T_case9	Array_struct_1	NO	YES	YES	YES
T_case10	Array_struct_2	YES	YES	YES	YES
FNR		**80%**	**40%**	**70%**	**20%**

First, the results of Splint and PML_Checker show a higher effectiveness compared with the other two tools in the experiment. Splint has a high effectiveness because it mainly checks the specifications in programs, which is sensitive to different data types. Splint has a high effectiveness because it simplifies the analysis for complex data types in the process of abstracting the control flows.

Second, comparing PML_Checker with Splint, Splint reported memory leaks on the latter three data types (linked list, struct and array_struct), except the memory leaks in an array in Fig. 8. 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 to Splint. For the former approaches, that is CppCheck and RL_Detector, there is a large space to improve in analyzing the complex data types.

RQ3: *Accuracy and Run time for Benchmarks*

The third research question is to compare the accuracy achieved by the 4 approaches, and the run time recorded by the corresponding 4 tools in terms of open benchmarks. There are two experiments.

Experiment I This experiment analyzed the test results and run time of the four tools on SPEC CPU 2000 and SIR, and the *FPR* was selected to measure the accuracy of different approaches on the same test suite. Table IV shows the test results on SPEC CPU 2000 and Table V shows the test results on SIR. Compare our approach with other approaches, we have the following observations.

First, in the view of *TW* from Table IV and Table V, Splint reported most bugs, and PML_Checker ranked second, while CppCheck reported the least bugs. However, in the view of *FP*, the *FPR* of Splint is the highest, and the *FPR* of the other three tools are acceptable.

Second, we analyzed the effectiveness of the CFG projection approach by comparing the MLF of test results. The table of test results is converted into MLF of test results by constructing a matrix. The results from Table IV can be abstracted to a sparse matrix consists of 15 quaternions. We compress 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.

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

TABLE IV
TEST RESULTS ON SPEC CPU 2000

	Size	A1		A2		A3		A4	
Program	Kloc	TW	FP	TW	FP	TW	FP	TW	FP
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
FPR		**11%**		**44%**		**9%**		**13%**	

TABLE V
TEST RESULTS ON SIR

	Size	A1		A2		A3		A4	
Program	Kloc	TW	FP	TW	FP	TW	FP	TW	FP
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
FPR		**19%**		**30%**		**8%**		**18%**	

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

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

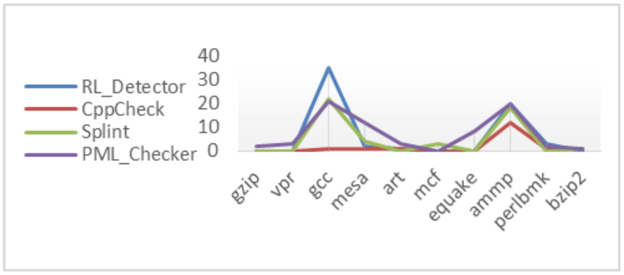

Fig. 9. The MLF of each tool on SPEC CPU 2000

In Fig. 9, the abscissa shows the name of the program corresponding to each row in the matrix, the ordinate shows

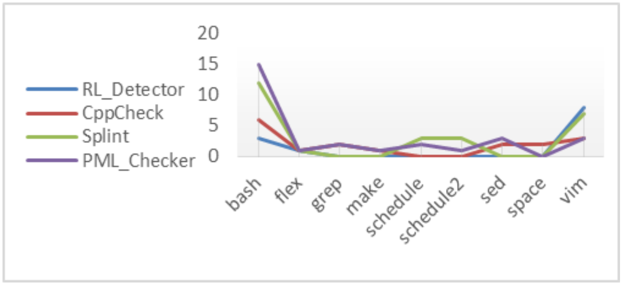

Fig. 10. The MLF of each tool on SIR

the value of corresponding MLF. This figure displays the number of memory leaks confirmed by manually checking, among the test results on the 15 programs from SPEC CPU 2000. We conclude from the figure that the CFG projection has a good detection result on all the test sets except gcc, for the reasons that the program gcc is larger than others in scale. Therefore, we deduce that the scalability of the CFG projection approach for large scale test objects needs to be improved. While in Fig. 10, due to the smaller amount of loc(line of code) 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.

Third, considering symbolic method are used in A1(CppCheck), A3(RL_Detector) and A4(PML_Checker), meanwhile, in order to verify whether our approach to influence the running time, so we chose SPEC CPU 2000 as the test object due to its large amount of code and large number of files, compared and analyzed run time of the three tools. Table VI lists the run time on this benchmark. 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 did not affect the run time of PML_Checker.

TABLE VI
RUN TIME ON SPEC CPU 2000

Program	**Size(Kloc)**	**A1(s)**	**A3(s)**	**A4(s)**
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

Experiment II This experiment analyzed the results of the four tools on test cases from SARD. The *FPR* and *FNR* were measured to the accuracy of different approaches on the same test suite. Table VII shows the test results on these test cases, and Fig. 11 is a histogram that compares test results metrics of the four detection tools. Comparing our approach with other approaches, we have the following observations.

The *FPR* of PML_Checker is the lowest among the four tools. For the analysis of *FPR*, there are no false positives in the results of RL_Detector, CppCheck and PML_Checker, while Splint reported a false positive, which confirmed that Splint has a high *FPR* in experiment I. This experiment failed to compare the *FPR* due to the small scale of the programs, but the test results on *FNR* reflected the accuracy of our approach in detecting memory leaks.

TABLE VII
TEST RESULTS ON SARD TEST CASES

Test Cases	**A1** (TW)	**A2** (TW)	**A3** (TW)	**A4** (TW)
bad cases(20)	10	4	6	14
good cases(20)	0	1	0	0
FPR	**0**	**5%**	**0**	**0**
FNR	**50%**	**80%**	**70%**	**30%**

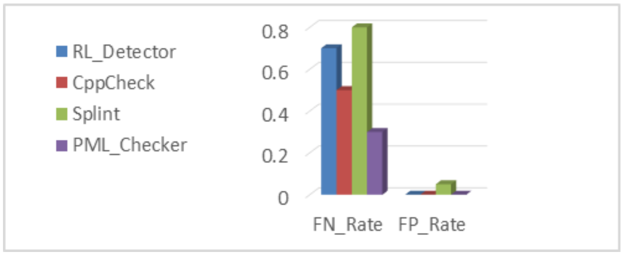

Fig. 11. The bar chart of test results

C. Summary

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

- For the complex control flows and complex data types analysis, our projection-based approach is better than the other three approaches.
- For the accuracy of detection, our projection-based approach shows a lower false negative rate compared to other tools, while it shows a high false positive rate in some public benchmarks (such as SPEC CPU 2000 and SIR). Therefore, in the next step, more detailed data flow analysis will be added into our approach.
- For the scalability of detection, our experiment needs to expand the range of detection. Specifically, we will detect the large scale benchmarks with millions of code lines, or source code from some open source software.

V. RELATED WORKS

A. Approaches on Static Detection

Currently, the common approaches [**?**] used in static memory detection include symbolic execution followed by constraint solving, type inference[9], rule-based inspection [**?**], besides those mentioned in the previous section. However,

[9]Type inference. https://en.wikipedia.org/wiki/Type_inference. 2017.

all these approaches have limitations when dealing with large scale programs. For symbolic execution, the number of paths will increase exponentially with the increase of the program size, which leads to problems such as "path-explosion" and "infinite-search-space", and thus increases the time complexity of entire detection. Type inference automatically derives the type of variables by tools, this approach is suitable for analyzing large scale programs, but it is not applicable to the control flow analysis. Compare to this approach, CFG based approach, that we are following, is more intuitive and accurate in detecting memory leaks. Regarding rule-based inspection, its scalability is poor due to that describing a program into rules is complicated and error-prone.

B. Tools on Static Detection

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 [**?**], 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[10] 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 [**?**] 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 [**?**], [**?**], 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/C++ programs, especially in the program with complex control flows.

C. Regarding the Complex Control Flow Concept

There are few studies on complex control flow in the field of static analysis. As in [**?**], this concept is mainly used in the process of dynamic testing to predict some frequently executed paths, in order to eliminate false positive caused by the evaluating the conditional branches. In [**?**], the concept is only used to show a method to measure the complexity of programs, which is irrelevant to memory detection.

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

D. Research on Projection in Program Analysis

In the field of program analysis, especially in the detection of program dynamic memory leaks, the researchers are more inclined to adopt the dynamic memory modeling methods. For example, [**?**] builds a heap abstraction model that restrains the heap model by combining multiple abstraction positions into a summary position, however in practical applications, there is still a great gap on accuracy between the results of this method and the exact results. [**?**] builds a pointer behavior model, this model utilizes model checking tools to verify the reachability of assertions to analyzing memory leaks. [**?**] builds a bounded model, where the memory leaks properties are added. This method transforms the memory leaks problem into solving satisfiability problem, thereby achieves the memory leaks detection. However, this detection scheme needs to reduce the generated verification space and accordingly improve the accuracy of verify. Our approach builds a program model, which is used to be traversed and stored into the stack following some rules for detecting memory leaks.

In recent years, the idea of projection is often applied to computer vision [**?**], mathematical model selection [**?**] 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. [**?**] analyzes the specification and abortion of programs by the projection of functions. While in terms of security, 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. [**?**] detects dead cycles in control flows and confusion in data flows by projecting the control flow graphs. In our approach, we redefine the projection of the program control flow graphs, and solve security problems with a specific algorithm for detecting memory leaks.

VI. Conclusion

In this paper, we focus on CFG based memory leaks detection with complex control flows. We propose a novel projection-based approach to increase the efficiency in detecting memory leaks. We develop a prototype system named PML_Checker and evaluate our approach by comparing with other approaches on public benchmarks (SARD and SPEC CPU 2000). Experimental results show that our approach has better effectiveness than other approaches in detecting memory leaks on programs with complex control flows and complex data types. In addition, our approach can accurately find memory leaks, i.e., finding more leaks with much fewer false negatives than previous detection tools. As future work, we plan to improve our approach in scalability, specifically, by combining other methods such as data flow analysis to improve the detection accuracy.