2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code

Gang Fan*, Rongxin Wu*§, Qingkai Shi*, Xiao Xiao†, Jinguo Zhou†, Charles Zhang*
*Hong Kong University of Science and Technology
{gfan, wurongxin, qshiaa, charlesz}@cse.ust.hk
†Sourcebrella Inc.
{xx, jinguo}@sbrella.com

***Abstract*—Detecting memory leak at industrial scale is still not well addressed, in spite of the tremendous effort from both industry and academia in the past decades. Existing work suffers from an unresolved paradox – a highly precise analysis limits its scalability and an imprecise one seriously hurts its precision or recall. In this work, we present SMOKE, a staged approach to resolve this paradox. In the first stage, instead of using a uniform precise analysis for all paths, we use a scalable but imprecise analysis to compute a succinct set of candidate memory leak paths. In the second stage, we leverage a more precise analysis to verify the feasibility of those candidates. The first stage is scalable, due to the design of a new sparse program representation, the use-flow graph (UFG), that models the problem as a polynomial-time state analysis. The second stage analysis is both precise and efficient, due to the smaller number of candidates and the design of a dedicated constraint solver. Experimental results show that SMOKE can finish checking industrial-sized projects, up to 8MLoC, in forty minutes with an average false positive rate of 24.4%. Besides, SMOKE is significantly faster than the state-of-the-art research techniques as well as the industrial tools, with the speedup ranging from 5.2X to 22.8X. In the twenty-nine mature and extensively checked benchmark projects, SMOKE has discovered thirty previously-unknown memory leaks which were confirmed by developers, and one even assigned a CVE ID.**

***Index Terms*—memory leak, static bug finding, use-flow graph, value-flow graph**

I. INTRODUCTION

Despite the tremendous research progress in recent decades [1]–[9], the detection of memory leaks in industrial-scale is still pretty much an unsolved problem. In the first half of the year 2018, more than 680 memory leak bugs have been reported in Firefox [10] and Chrome [11]. More than 240 CVE (Common Vulnerabilities and Exposures) entries in 2017 are memory leaks bugs [12]. Apparently, with the explosive growth of the code size and the complexity in modern software [13], a practical memory detector needs to be *highly scalable*, checking millions of lines of code within minutes , *and precise*, understanding complex path conditions with less than 30% false positives [14], [15].

The state-of-the-art approaches suffer from the scalability and precision paradox. One category of the approaches [4]–[9] give up path sensitivity for scalability, inevitably introducing imprecise results. For example, we observed that, SABER [9], a recent path-insensitive memory leak detection technique, incurs a false positive rate of 66.7% in our evaluation. Another category [1]–[3] traverse the control flow graph and use the path-sensitive analysis to achieve high precision. However, they are known to easily suffer from scalability issues, especially for the whole-program analysis. For example, SATURN [3] is reported to have spent more than 23 hours in checking memory leaks for a 5MLoC code base. Our experiment shows that CSA [1] and INFER [2] fail to analyze large projects of over 2MLoC in two hours.

Our idea to resolve this paradox is based on an observation that, in real programs, only a small proportion of program paths lead to memory leaks. Therefore, instead of using a sledge hammer, i.e., the expensive path-sensitive analysis, for all paths, we use a two-staged analysis by first computing a succinct set of candidate memory leak paths through a novel scalable and path-insensitive method, followed by a more precise and heavy-weight verification of the feasibility of these paths, in order to achieve path-sensitivity.

More specifically, to check millions of lines of code in minutes, we believe that the sparse value-flow analysis, already widely adopted in finding memory leaks [8], [9], is the right direction as it tracks values along the data dependence relations on the value flow graph (VFG) instead of the control flow graph, skipping irrelevant program statements to achieve scalability. However, we observe that the VFG, originally intended for program transformations [16], is not suitable for arbitrary finite-state-machine properties such as the memory leak problem, due to the omission of flow information(order of events) [8]. Therefore, instead of finding a leak path, VFG-based methods need to deduce the leak path by checking whether the non-leak paths (i.e., paths where the heap object is safely freed) cover all control flow paths from where the heap memory is allocated. Such an analysis is equivalent to solving a $k-$SAT problem, which is *NP*-hard with the input size $k > 2$ (k represents the number of the branch conditions) [17]. This induces a very high time complexity in theory and may greatly compromise the efficiency and scalability in practice. For instance, the most recent VFG technique, Pinpoint [18], in spite of its leap in achieving scalability and precision, still cannot complete the analysis in some large subjects in our

§Rongxin Wu is the corresponding author.

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00025

experiments.

To overcome this limitation, we designed an extension of VFG, namely the use-flow graph (UFG), that encodes not just the definition but also the use of the problem-relevant heap objects. All use sites of the same heap object in UFG are ordered according to the control flow, to check finite-state-machine properties and to use polynomial-time graph search methods to find possible memory leak paths. Path-sensitivity is achieved in the second stage by using a dedicated constraint solver to verify the feasibility of these candidate leak paths. The verification process is efficient because, at this stage, the number of paths is very small (only 21 on average) and we use a customized constraint solver to further filter out "easy-to-contradict" ones, leading to a further pruning of the paths. Finally, we invoke a full SMT solver, such as Z3, for the remaining candidates.

To evaluate the scalability and effectiveness of our proposed technique, we implemented a tool, SMOKE, and applied it to the SPEC2000 benchmark programs and seventeen well maintained open source projects. The experimental results have demonstrated that SMOKE is highly efficient and effective, as it could finish checking industrial-sized projects, up to 8MLoC, in forty minutes with an overall false positive rate of 24.4%. This is aligned with the common industrial requirements of checking millions-of-LoC code [14], [15]. In the twenty-nine mature and already extensively checked third-party benchmark projects, SMOKE found thirty previously unknown memory leaks, all confirmed by the original developers. One of the reported leaks was even assigned with a CVE ID due to its high severity.

We highlight our contributions as follows:

- We present the design and the implementation of SMOKE, a staged approach for detecting memory leaks. SMOKE is faster, more scalable, and more precise than the state-of-the-art approaches.
- We present a novel sparse program representation, namely the use-flow graph, which allows us to efficiently and effectively detect memory leaks. Using the use-flow graph, we can model the memory leak detection problem as a *P* problem rather than an *NP*-hard problem on value-flow graph.
- We extensively evaluated SMOKE with standard benchmarks and a broad spectrum of open source projects. The experimental results demonstrate that SMOKE achieves the speedup ranging from 5.2X to 22.8X, compared with the state-of-the-art techniques.

This paper is organized as follows. We first present motivating examples in Section II. Section III describes our approach. The implementation and evaluation are presented in Section IV and Section V, respectively. Section VI discusses related works and this paper is concluded in Section VII.

II. MOTIVATING EXAMPLES

In this section, we use three examples to illustrate the limitations of the conventional Sparse Value-Flow Analysis (SVFA) [8], [9] for detecting memory leaks, which gives a better understanding of the key insights of our approach.

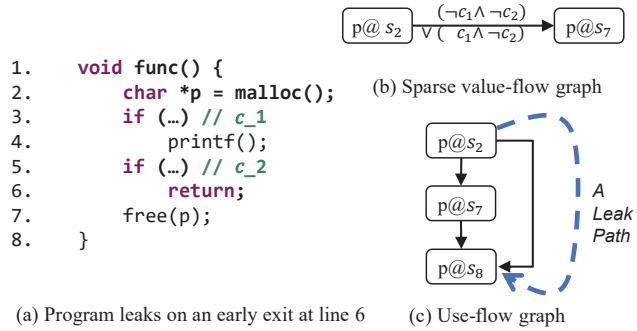
A. Reducing Complexity with Use-Flow Graph

Conventional SVFA techniques, such as FASTCHECK [8], start with the sparse value-flow graphs (SVFG) as shown in Figure 1(b) and Figure 2(b) for the code snippets in Figure 1(a) and Figure 2(a), respectively. In those SVFGs, each edge represents a data-dependence relation, denoting the flow of value. For example, the value-flow edge $p@s_2 \rightarrow p@s_7$ in Figure 1(b) implies that the allocated heap object pointed to by p *may* be released at the statement $s7$ (we use s_i to represent the statement at line i). In order to decide if the memory is always released properly, FASTCHECK annotates the control-flow conditions on the value-flow edges and relies on a constraint solver to solve the conditions. For the example in Figure 1, we will solve the condition $F_{leak} = \neg((\neg c_1 \wedge \neg c_2) \vee (c_1 \wedge \neg c_2))$. If F_{leak} is satisfiable, there exists a path with no memory release operations, leading to a memory-leak vulnerability. Note that, in general, solving F_{leak} is a $k-$SAT problem [17] (k represents the number of the branch conditions), which has the best time complexity of $O((2 - \frac{2}{k+1})^n)$ [19] and is proved to be *NP*-hard [17] for $k > 2$.

In SMOKE, we propose a new type of sparse program representation, named use-flow graph (UFG), that still contains the necessary control-flow information. Specifically, the UFG encodes the definition and the use of problem-relevant heap objects, which is similar to the sparseness feature in VFG. In addition, it records the control flow order of uses of the same heap object, which enables the checking of finite-state-machine properties such as memory leaks. Figure 1(c) shows the UFG of the code in Figure 1(a). The UFG slices away the unnecessary program statements, such as the ones in Lines 3 and 4, and only encodes the necessary data dependence and the control dependence. Meanwhile, UFG explicitly models the life cycle of a value by creating an *out-of-scope* node (i.e., $p@s_8$ in Figure 1(c)), which indicates that the heap object pointed to by p is no longer referenced. A simple graph traversal on the UFG shall discover that there exits a path where the pointer p is never "freed". Similarly, a simple graph traversal on the UFG in Figure 2(c) will verify the absence of memory leaks, because each path in the graph contains a memory release operation. It is noteworthy that, using UFG, we can obtain the same results with the same precision as that of FASTCHECK, without collecting constraints for each path and using a heavyweight constraint solver.

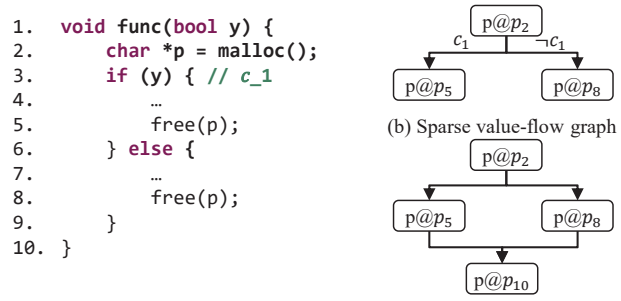
B. Regaining Path-Sensitivity via Staged Constraint Solving

Since most of the allocated memory spaces are managed properly in common software, many cases of memory leaks can be detected by graph traversals on the UFG with the flow-sensitive precision. However, there still exists a handful of cases that require capturing branch correlations, the importance of which is illustrated by Figure 3. This example is leak-free because the memory is allocated and released under the



(a) Program leaks on an early exit at line 6 (b) Sparse value-flow graph (c) Use-flow graph

Fig. 1. A memory leak example.



(a) A code snippet contains no memory leaks (b) Sparse value-flow graph (c) Use-flow graph

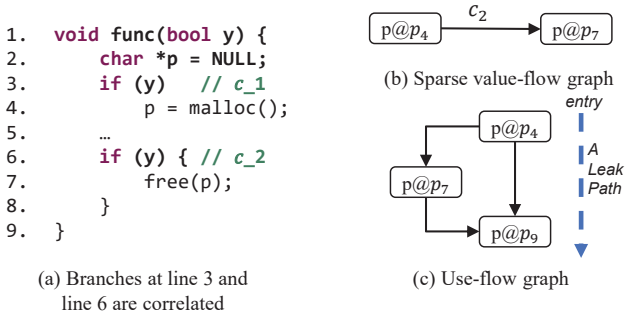
Fig. 2. An example without any memory leaks.

same condition at Line 4 and Line 7, respectively. However, FASTCHECK, as well as other existing SVFA techniques (e.g., SABER), will mistakenly report it as a memory leak due to the negligence of the branch correlation. Such negligence is caused by the large overhead of collecting path conditions for many paths and the high complexity of conventional constraint-solving methods.

To correctly identify that the code snippet is leak-free, SMOKE employs a staged constraint solving process to verify the feasibility of the candidate leak paths, such as the one in Figure 3(c). First, we adopt a linear-time solver to filter obvious infeasible paths. Most of the false warnings involving branch correlations can be pruned at this stage. For the remaining paths with complex path conditions, we adopt a fully-featured SMT solver, such as Z3 [20], to check their feasibility to achieve a low false positive rate.

III. MEMORY LEAK DETECTION

In order to speed up the memory leak detection without losing the precision, we make two design decisions. First, we use a lightweight finite-state analysis with a new sparse program representation, the use-flow graph (Section III-A). Second, we achieve the precision of path-sensitivity through a dedicated constraint solver (Section III-B). In this section, we assume that a program consists of functions in SSA form and the pointer relations in the program have been resolved. We discuss how to satisfy the assumptions in our implementation in the next section.



(a) Branches at line 3 and line 6 are correlated (b) Sparse value-flow graph (c) Use-flow graph

Fig. 3. A false memory leak due to an infeasible program path.

A. Finite State Analysis with Use Flow Graph

Our analysis is based on a new type of sparse program representation, the use-flow graph (UFG). Compared to the conventional sparse value-flow graph, UFGs have the following features for checking the finite-state properties of a program:

- It models the whole life cycle of program variables by encoding the property-related control flows, which we referred to as use flows.
- It can be efficiently built in linear time with regard to the program size.

A finite-state property of a program can be modeled as a finite-state machine (FSM), which defines the valid sequences of operations that can be performed upon an object.

Definition 1 (Finite-State Machine (FSM))**.** A finite-state machine of a program property is a quintuple $M = (\Sigma, S, s_0, \delta, F)$, where

- Σ is a finite, non-empty set of classes of program points.
- S is a finite, non-empty set of the states of a program object.
- $s_0 \in S$ is the initial state.
- δ is the state-transition function: $\delta : S \times \Sigma \mapsto S$. $\delta((s_1, P)) = s_2$ means that the state s_1 of an object can transit to the state s_2 if the object goes through a program point $p \in P$.
- $F \subseteq S$ is the set of final states.

Example 1. As shown in Figure 4, we use an FSM to model different states and state transitions of a heap object. There are four states in the FSM: Allocated (**A**), Freed (**F**), Error (**E**), and eXit (**X**). **A** is the initial state, and both **E** and **X** are the final states. **E** indicates a safety violation of the finite-state property, such as memory leak and double free, and **X** denotes a normal exit. The state **A** will transit to the state **F** if a newly-allocated heap object goes through a program point where a "free" operation is performed. The state **A** will transit to the state **E** (i.e., a memory leak issue) if a heap object goes out of its life scope and can be no longer referenced.

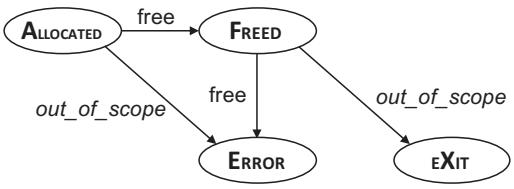


Fig. 4. The finite-state machine for a heap object.

TABLE I
THE ANALYSIS STEPS IN EXAMPLE 3.

Step	Vertex	Before	After
1	$t@s_2$	-	{**A**}
2	$t@s_3$	{**A**}	{**A**, **F**}
3	$p@s_{10}$	{**A**}	{**A**}
4	$p@s_{11}$	{**A**}	{**F**}
5	$p@s_{12}$	{**A**}, {**F**}	{**A**, **F**}
6	$t@s_6$	{**A**, **F**}	{**F**, **E**}
7	$t@s_7$	{**A**, **F**}, {**F**}	{**A**, **F**}
8	$t@s_9$	{**A**, **F**}	{**E**, **X**}

Given an FSM, we can build the sparse program representation, the use-flow graph, as follows.

Definition 2 (Use-Flow Graph (UFG))**.** The use-flow graph of a program with regard to a given FSM is a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges:

- A vertex $o@p \in V$ represents an object o at a program point p. p is one of the following program points:
  - $p \in P$ where $P \in \Sigma$ and $\delta(s, P) \neq s$. That is, p is a program point that can cause a transition between different states.
  - p is where we call a function using o as an actual parameter.
  - p is the entry of a function where o is a formal parameter.
  - for each $o@p' \in V$, p is the dominance frontier [21] of p'. Intuitively, the dominance frontier is where the states of the object o from different paths can be merged.
- A directed edge $(o_1@p_1, o_2@p_2) \in E$ if and only if o_1 and o_2 represent the same object and there exists a control flow path from p_1 to p_2. The path does not go through any other program point p_3 such that there exists o_3 representing the same object and $o_3@p_3 \in V$.

Given a program and the FSM of a property, the UFG can be built efficiently according to Definition 2 in polynomial time. The basic idea is to traverse the control flow graph of each function and remove the statements at program points that are irrelevant according to the FSM.

Example 2. Figure 5 illustrates an example of the UFG w.r.t. the FSM in Figure 4. As a sparse representation, all irrelevant program statements such as the ones at Lines 4 and 5 are not modeled. We have the vertices $t@s_7$ and $p@s_{12}$ because they

```
1.  void foo() {
2.      int *t = malloc(…);
3.      bar(t);
4.      int q = qux();
5.      printf("%d\n", q);
6.      if (…) free(t);
7.      …
8.      return;
9.  }
```

```
10. void bar(int *p) {
11.     if (…) free(p);
12. }
```

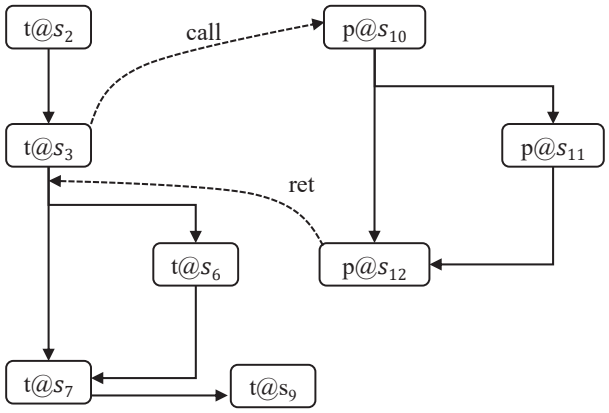


Fig. 5. An example to illustrate UFG construction. s_i represents the program point at Line i.

are the dominance frontiers of $t@s_6$ and $p@s_{11}$, respectively. At the program points s_7 and s_{10}, the states of the heap object from different program paths can be merged. We have the vertex $t@s_9$ because this is the end point of the scope of the heap object t.

Next, we describe how to detect memory leaks using UFG. The basic idea is to check the state of a heap object by traversing the UFG inter-procedurally. When traversing a UFG, we use a set to keep track of the states at each vertex. From the memory allocation site, the state **A** is added to the set. The state set is propagated forward along the UFG edges. When visiting a vertex corresponding to a transition of the FSM, we transit a state accordingly. Otherwise, the states remain unchanged. At each merge point, we merge the state sets from different paths by the set union operation. Instead of giving a complex formal representation of the algorithm, we use the following example to illustrate the process.

Example 3. We use Table I to illustrate the steps of our state analysis on the UFG in Figure 5. Each row of the table shows the state set before and after a vertex. For example, in Steps 5 and 7, since the program points, s_{12} and s_7, are dominance frontiers where two state sets from different paths meet, we use the set-union operation to merge the states. In Step 8, since the program point s_9 represents the end of the scope of the heap object t, the state **A** transits to the state **E** and the state **F** transits to the state **X**. Since the state **E** is obtained via an *out-of-scope* operation, we report a memory-leak candidate. This candidate will be verified path-sensitively as detailed later.

As illustrated by the above example, the UFG is traversed inter-procedurally. The context-sensitivity is achieved via the CFL-reachability method [22]. That is, we assign a string to each state during the graph traversal to check the validity of the context. When propagating a state along a call edge at a call site cs, we append a left parenthesis $(_{cs}$ to the string. When propagating a state back to a call site cs along a return edge, we append a right parenthesis $)_{cs}$ to the string. A state propagation is valid only if the string has matched parentheses.

We create a function summary for each function in a demand-driven way. That is, when reaching a call site, we check if the callee has a usable summary so that we do not need to reanalyze the callee. Otherwise, we create a summary after the callee is analyzed. In our approach, a summary is a map between the input states of a parameter p and the states of p at the end of a function. Therefore, it is unnecessary to repeat the analysis of a function at different call sites if its summary has been created.

Our analysis can be efficiently implemented using the RHS algorithm [22]. Given an FSM $M = (\Sigma, S, s_0, \delta, F)$ and its corresponding UFG $G = (V, E)$, the time complexity of the algorithm is $O(|V|^2|S|(|E||S| + |Calls||S|^2))$, where $|Calls|$ is the number of call sites in the program to check.

B. Staged Path-Sensitive Verification

```
1.  void foo(bool c) {
2.       int *t = malloc(…);
3.       if (c)
4.            free(t);
5.       …
6.       if (!c)
7.            free(t);
8.       …
9.  }
```

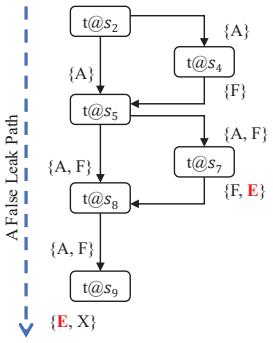


Fig. 6. A false memory leak case whose path condition contains an apparent contradiction $c \wedge \neg c$.

Some of the UFG paths generated by the aforementioned state analysis are infeasible, which may result in false positives as illustrated in Figure 6. To reduce the false positives, we introduce a path-sensitive verification step where we collect and solve path conditions of each memory-leak candidate path. Although the path-sensitive verification is usually expensive, our approach can be efficient because of two important observations. First, we observe that the state analysis only produces a handful of memory leak candidates that require a path-sensitive verification. This is because most of the heap memory spaces are managed safely in practice. Second, we observe that the constraint in a path condition usually has apparent contradictions such as $a \wedge \neg a$, as shown in the example in Figure 6. The reason is that the programmers tend to use direct contradictions to ensure some required logical properties. Thus, we first adopt a linear-time solver to detect apparent contradictions and to filter obviously infeasible paths. For the remaining complex cases, we use a fully-featured SMT solver, such as Z3 [20], to check the path feasibility. The basic idea of the linear-time solver is similar to the one used in prior study [18], and it continuously collects the sets of positive and negative atomic constraints during the construction of a path condition. An atomic constraint is a first-order logic formula that does not contain any logic operator like $\wedge$, $\vee$, and $\neg$. For example, $a < 2b$ and c are two atomic constraints in $a < 2b \wedge c$. If a path condition has an atomic constraint a in both sets, this path condition must contain $a \wedge \neg a$ and, thus, is unsatisfiable.

IV. Implementation

We have implemented SMOKE on top of the LLVM framework [23], which takes an LLVM-Bitcode file as input, for detecting C/C++ memory leaks.

Similar to some earlier work [8], [9], [24], we have some unsound trade-offs to make the detector more practical. We assume a path in UFG is safe if this path flows to a global pointer or a container (e.g., std::list, std::vector). We do not report memory leaks if a path ends in a call to $exit()$. If a heap object is used as an argument of a library function (i.e., a function that is not defined in the bitcode) on a path, we conservatively treat it as an unknown value and stop searching this path. We manually modeled some common library functions, such as $memcpy$ and $memset$, to improve the precision and the recall. Similar to FASTCHECK [8], we do not check the arithmetic operations on heap pointer, $free(p + y)$ is simply treated as $free(p)$.

Figure 7 illustrates the overview of our tool SMOKE. SMOKE has four phases: Pre-Analyses, UFG Construction, State Analysis and Path-Sensitive Verification. Here we only discuss the details of the Pre-Analyses phase since other three phases have already been discussed in Section III.

In the pre-analyses phase, we conduct several analyses to compute the necessary information for later phases. We first use a flow-sensitive and context-sensitive pointer analysis similar to the one used in PINPOINT [18] to compute the data dependence. Control dependence is computed on demand, since not all functions in a program are related to memory leak detection. We construct the control dependence by computing the dominance frontier in the reverse graph of the control flow graph [21]. To construct the call graph, we use the must-alias results from the pointer analysis to resolve function pointers, and adopt a class hierarchy analysis [25] to resolve virtual function calls.

During the state analysis, we only consider the case when the memory allocation operation is successful. For instance, if a heap object is created in $x = malloc()$, we do not consider the case when the test $x == NULL$ is true. We use a light-weight data-flow analysis to identify and ignore UFG edges on which heap memories have not been successfully allocated.

V. Evaluation

We evaluate the precision, the recall and the scalability of SMOKE by comparing four well-known static analysis tools

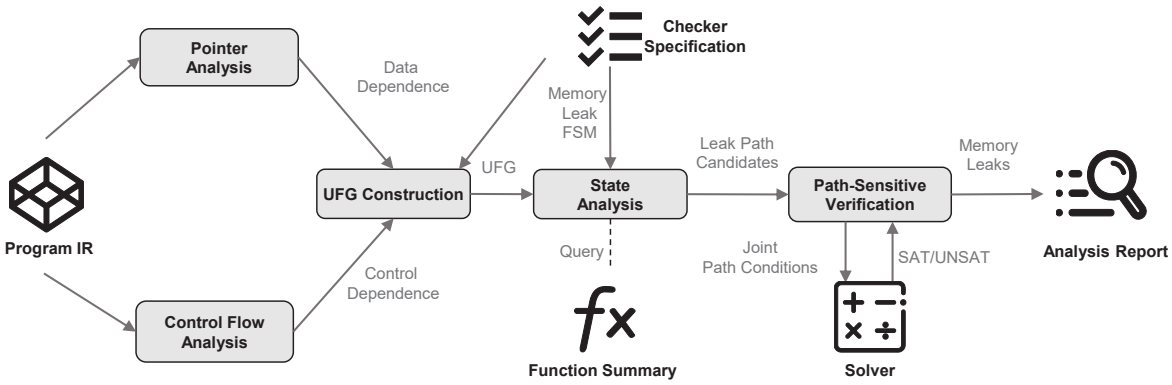


Fig. 7. System overview of SMOKE

that have memory leak detectors, from both academia and industry. We compare to both SABER [9] and PINPOINT [18], since they are the state-of-the-art SVFA based approaches with the precision of flow-sensitivity and path-sensitivity, respectively. We also choose CSA [1] and INFER [2], two prominent and mature open-source tools from the industry. We plan to evaluate other memory leak detection tools, such as SATURN [3] and CALYSTO [26]. However, they are either publicly unavailable or outdated for running in the environments we are able to set up. To demonstrate the usefulness of our approach, we also seek confirmations from original developers of the subjects we use.

For each baseline, we only enable the memory leak detector, since some tools may have other bug detectors. We set the timeout to 2 hours, and configure each tool with its default settings. All the experiments are performed on a moderate computer running Ubuntu-16.04 with an Intel Core i5-6500 quad-core processor, and 64GB physical memory.

A. Subjects for Evaluation

We evaluate SMOKE, SABER, PINPOINT, CSA and INFER using twenty-nine subjects, including the twelve benchmark programs from standard SPEC CINT2000 [27], a commonly used benchmark in the existing literature [9], and seventeen well tested open-source projects. Table II shows the basic information of the evaluation subjects. The size of these subjects ranges from a few thousand of lines of code to nearly eight million of lines of code. These projects are widely used and well tested before release. Some of them are regularly scanned by free or commercial static tools such as COVERITY SAVE* and, thus, expected to have high code quality. We divide the subjects into two categories: medium-sized projects (including all SPEC CINT2000 programs and the projects with code size less than 1MLoC) and large-sized projects with code size larger than 1MLoC.

*https://scan.coverity.com/projects/

B. Scalability

To evaluate the scalability of each tool, we first check whether it can successfully analyze the selected subjects under our experimental environment (which is a typical desktop computer rather than a powerful cluster) and within the time budget (2 hours). As shown in Table II, SMOKE successfully analyzes all twenty-nine projects in forty minutes, while others fail to analyze some of the projects due to either crash (out-of-memory or segmentation faults) or timeout. For example, SABER, having the lowest success rate, fails on nine projects, including two medium-sized and seven large-sized projects. PINPOINT and CSA achieve the second-best success rate, but still fail on four projects, which are mostly large-sized projects.

We further compare the time cost of analysis of all the tools shown in Table II. On average, SMOKE achieves the highest efficiency and is significantly (>5.2X faster than SABER, >13.0X faster than PINPOINT, >12.4X faster than CSA, >22.8X faster than INFER) faster than other tools. SMOKE performs better than SABER and PINPOINT in all subjects. Compared to CSA and INFER, SMOKE also performs better on all the subjects, except one medium-sized project, *253.perlbmk*.

Overall, the evaluation results show that, compared to other tools, SMOKE achieves the highest scalability and scales to millions of lines of code without requiring too much computation resource (using a desktop computer).

C. Precision

Following the common practice in the literature on memory leak detection [8], [9], we manually check each error report provided by all the tools, and then compare their false positive rate (**FP rate** for short) to evaluate the precision. This process may be subjective and introduce threats to the validity of the FP results. Therefore, we ask three software engineers to cross-validate the results and, meanwhile, seek confirmations from the developers of the subjects. In addition, we release the reports and data for inspection.†

†https://smokeml.github.io/data

TABLE II
ANALYSIS TIME COMPARISON

Origin	Project	SLOC (kLoC)	#Allocations Sites	# Free Sites	# Funcs	SMOKE	Saber		PINPOINT		CSA		Infer	
						Time (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
SPEC CINT2000	164.gzip	6	14	3	89	0.8	0.8	1.0X	6.2	7.5X	8.8	10.7X	5.4	6.6X
	175.vpr	11	3	171	300	2.8	2.8	1.0X	20.4	7.3X	46.4	16.6X	74.7	26.7X
	176.gcc	133	96	73	2,218	33.2	97.6	2.9X	1154.4	34.8X	335.6	10.1X	Crash	N/A
	181.mcf	2	4	2	26	0.07	0.3	4.6X	1.8	25.7X	9.2	130.7X	6.0	85.0X
	186.crafty	8	0	0	42	0.11	1.07	9.7X	6.4	58.5X	13.2	119.6X	14.8	134.9X
	197.parser	8	148	129	324	1.9	2.4	1.3X	68.3	36.5X	46.2	24.7X	45.4	24.3X
	252.eon	22	142	23	3,367	8.2	9.8	1.2X	77.8	9.5X	50.8	6.2X	176.0	21.4X
	253.perlbmk	8	17	396	1,069	117.7	127.4	1.1X	2258.0	19.2X	28.6	0.2X	89.2	0.8X
	254.gap	36	1	1	843	0.7	43.3	61.0X	192.0	270.4X	154.4	217.4X	122.9	173.2X
	255.vortex	49	11	3	923	1.1	17.4	15.3X	76.0	66.7X	65.6	57.5X	483.6	424.2X
	256.bzip2	3	11	8	74	0.09	0.4	4.7X	4.7	51.7X	8.2	90.6X	13.4	148.9X
	300.twolf	18	162	62	191	0.35	6.0	17.1X	36.1	103.1X	9.7	27.6X	47.6	135.9X
(<1MLoC) Open Source Projects	Bftpd	5	87	105	142	0.7	0.7	1.0X	2.9	4.2X	9.7	14.3X	18.0	26.6X
	Htop	15	12	230	376	4.6	5.3	1.2X	14.0	3.0X	19.8	4.3X	54.1	11.7X
	Memcached	17	59	102	350	2.7	3.2	1.2X	50.0	18.7X	25.4	9.5X	58.2	21.8X
	Caffe	34	910	633	13,742	71.2	2368.6	33.2X	511.6	7.2X	Crash	N/A	291.7	4.1X
	LAME	42	34	49	758	14.0	OOT	> 514.7X	36.4	2.6X	47.2	3.4X	120.8	8.6X
	Zlib	42	1	4	19	0.03	0.17	5.7X	0.54	18.0X	25.2	839.7X	45.3	1510.0X
	Tmux	42	108	885	1,293	19.5	66.6	3.4X	117.0	6.0X	141.9	7.3X	337.0	17.3X
	Apache httpd	179	10	23	1,178	11.5	12.4	1.1X	64.1	5.6X	333.1	29.0X	666.0	58.1X
	OpenSSL	300	24	5,649	12,157	261.9	OOT	> 27.5X	887.4	3.4X	359.8	1.4X	1179.2	4.5X
(>1MLoC) Open Source Projects	FFmpeg	1,001	3	1,707	74,260	174.1	OOT	> 41.4X	6767.2	38.9X	1998.3	11.5X	6563.5	37.7X
	Godot Engine	1,191	666	7,299	113,039	1698.0	OOT	> 4.2X	OOT	> 4.2X	3953.9	2.3X	OOT	> 4.2X
	MySQL	1,198	269	3,957	24,816	281.0	OOT	> 25.6X	1679.5	6.0X	1653.8	5.9X	6535.3	23.3X
	Chrome V8	1,201	6,152	18,906	260,401	2584.0	OOT	> 2.8X	OOT	> 2.8X	OOT	> 2.8X	OOT	> 2.8X
	Skia	1,233	6,670	22,587	215,678	394.0	OOT	> 18.3X	5639.0	14.3X	4330.3	11.0X	OOT	> 18.3X
	Blender	1,466	4,638	23,838	201,468	2034.6	OOT	> 3.5X	OOM	> 3.5X	2784.1	1.4X	OOT	> 3.5X
	Wine	4,108	2,011	6,645	133,055	264.0	362.0	1.4X	7142.9	27.1X	Crash	N/A	OOT	> 27.3X
	Firefox	7,998	21,019	42,291	666,187	2396.0	OOT	> 3.0X	OOM	> 3.0X	OOT	> 3.0X	OOT	> 3.0X
	Sum:	20,376	43,282	135,781	1,728,385	Geometric Mean:		> 5.2X		> 13.0X		> 12.4X		> 22.8X

OOT : out of time OOM: out of memory N/A : not available

Table III shows the comparison results. On average, SMOKE achieves an FP rate of 24.4%, which is the lowest one among all the tools. As discussed in Section V-B, not all the subjects can be successfully analyzed by all the tools. And the number of false positives in the failure cases (e.g., SABER fails in the project LAME) is counted as 0. For the medium-sized subject, the small number of failure cases (only 2 by SABER, 1 by CSA and INFER) allows a fair comparison of all the tools, where SMOKE outperforms all other tools with a lowest FP rate of 15.0%. The second best tool, PINPOINT, achieves a much higher FP rate of 41.7%. INFER performs the worst, with an FP rate of 87.5%. For the large-sized subjects, we also observe SMOKE achieves the lowest FP rate, which is consistent with the results in medium-sized subjects.

We further investigate why SMOKE can achieve the lowest FP rate. SABER does not capture the path correlation. Therefore, it cannot filter out cases with infeasible paths. Since PINPOINT and CSA achieve the precision of path-sensitivity, they report the smallest number of false positives. However, they report much fewer true positive instances, incurring the higher FP rate. INFER has the highest FP rate. A manual check reveals that INFER generates a specification for each analyzed function, and discards some of these specifications for unknown reasons. The omitted specifications produce imprecision in the whole program analysis. For example, a typical false positive case happens when the specification for a wrapper function of $free$ (i.e., a library function to free heap object) is discarded.

D. Recall

From the results in Table III, we observe that SMOKE reports more memory leaks than other tools. SMOKE reports 158 real memory leaks while the other four tools report only 48 real memory leaks in total.

SABER and PINPOINT make significant precision trade-offs to achieve efficiency. For instance, they limit the calling depth for inter-procedural analysis and only report memory leaks with six levels of function calls. They also stop analysis whenever they find that a heap object is assigned to a global variable on a path, even when the heap object obviously leaks on another path. For example, Figure 8 shows a leak found by SMOKE: A heap object allocated in *get_dll_name* flows to *dll_name*. It leaks when taking the true branch at line 207.

TABLE III
MEMORY LEAK RESULTS

Origin	Project	SMOKE	Saber	PINPOINT	CSA	Infer
		#FP/#Rep	#FP/#Rep	#FP/#Rep	#FP/#Rep	#FP/#Rep
SPEC CINT2000	164.gzip	0 / 1	1 / 2	0 / 0	0 / 0	0 / 0
	175.vpr	0 / 2	0 / 0	0 / 0	1 / 1	0 / 0
	176.gcc	0 / 8	3 / 7	0 / 3	0 / 1	N/A
	181.mcf	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
	186.crafty	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
	197.parser	0 / 2	0 / 0	0 / 0	0 / 0	0 / 0
	252.eon	1 / 4	0 / 1	0 / 2	2 / 3	0 / 0
	253.perlbmk	0 / 1	0 / 1	0 / 0	0 / 0	3 / 3
	254.gap	0 / 0	1 / 1	0 / 0	0 / 0	1 / 1
	255.vortex	0 / 1	0 / 2	1 / 1	0 / 0	0 / 0
	256.bzip2	0 / 1	0 / 1	0 / 0	0 / 0	0 / 0
	300.twolf	0 / 0	1 / 3	0 / 0	0 / 0	0 / 0
(<1MLoC) Open Source Projects	Bftpd	0 / 1	1 / 1	0 / 0	0 / 1	0 / 1
	Htop	0 / 1	0 / 0	0 / 0	0 / 0	3 / 3
	Memcached	1 / 2	3 / 4	0 / 1	0 / 0	2 / 3
	Caffe	0 / 0	0 / 0	1 / 1	N/A	0 / 0
	LAME	0 / 0	N/A	0 / 0	0 / 0	1 / 1
	Zlib	0 / 0	1 / 1	0 / 0	0 / 0	1 / 2
	Tmux	2 / 9	6 / 6	2 / 2	0 / 0	17 / 18
	Apache httpd	0 / 1	4 / 5	0 / 1	1 / 2	5 / 6
	OpenSSL	2 / 6	N/A	1 / 1	0 / 0	9 / 10
	Total:	6 / 40	21 / 35	5 / 12	4 / 8	42 / 48
<1MLoc	**FP Rate:**	15.0%	60.0%	41.7%	50.0%	87.5%
(>1MLoC) Open Source Projects	FFmpeg	1 / 1	N/A	0 / 0	1 / 1	17 / 17
	Godot	3 / 9	N/A	1 / 1	0 / 0	N/A
	MySQL	3 / 17	N/A	2 / 2	3 / 8	20 / 21
	Chrome V8	5 / 11	N/A	N/A	N/A	N/A
	Skia	5 / 11	N/A	1 / 1	0 / 2	N/A
	Blender	4 / 10	N/A	N/A	3 / 3	N/A
	Wine	1 / 21	19 / 25	2 / 5	N/A	N/A
	Firefox	23 / 89	N/A	N/A	N/A	N/A
	Total:	51 / 209	40 / 60	11 / 21	11 / 22	79 / 86
>1MLoC	**FP Rate**	26.6%	76.0%	66.7%	50.0%	97.4%
All	**FP Rate**	24.4%	66.7%	52.4%	50.0%	91.9%

SABER and PINPOINT cannot find this leak since it can flow to a global variable (*dll_delayed* or *dll_imports*) when taking the false branch at line 207.

CSA does not analyze function calls across different files. INFER can only handle a small number of function calls across different files. However, we observe that it is very common that a memory leak relates to functions from different files. As a result, CSA and INFER miss many real memory leaks. For example, Figure 9 shows a memory leak in Bftpd reported by SMOKE but not by CSA or INFER. A heap object allocated in a source file *cmd.c* leaks in another file *commands.c*.

E. Contribution of the Analysis Stages

To better understand the effectiveness and the efficiency of the staged analysis, we report the number of pruned candidates and the time cost of each stage in Table IV. Note that, since it is difficult to calculate the total number of the candidate paths starting from each allocation sites (e.g., path explosion due to loops), we estimate the lower bound of the total number of candidates using the number of allocation sites.

Location:import.c:277

```
195. /* add a dll to the list of imports */
196. void add_import_dll( const char *name, const char *filename )
197. {
198.     DLLSPEC *spec;
199.     char *dll_name = get_dll_name( name, filename );
200.     struct import *imp = xmalloc( sizeof(*imp) );
201.
202.     memset( imp, 0, sizeof(*imp) );
203.
204.     if (filename) imp->full_name = xstrdup( filename );
205.     else imp->full_name = find_library( name );
206.
207.     if (!(spec = read_import_lib( imp )))
208.     {
209.         free_imports( imp );
210.         return;
211.     }
212.
213.     imp->dll_name = spec->file_name ? spec->file_name : dll_name;
214.     imp->c_name = make_c_identifier( imp->dll_name );
215.
216.     if (is_delayed_import( dll_name ))
217.         list_add_tail( &dll_delayed, &imp->entry );
218.     else
219.         list_add_tail( &dll_imports, &imp->entry );
220. }
```

A memory is allocated and assigned to dll_name

Take the true branch at line 207

dll_name is leaked

Fig. 8. A memory leak in Wine

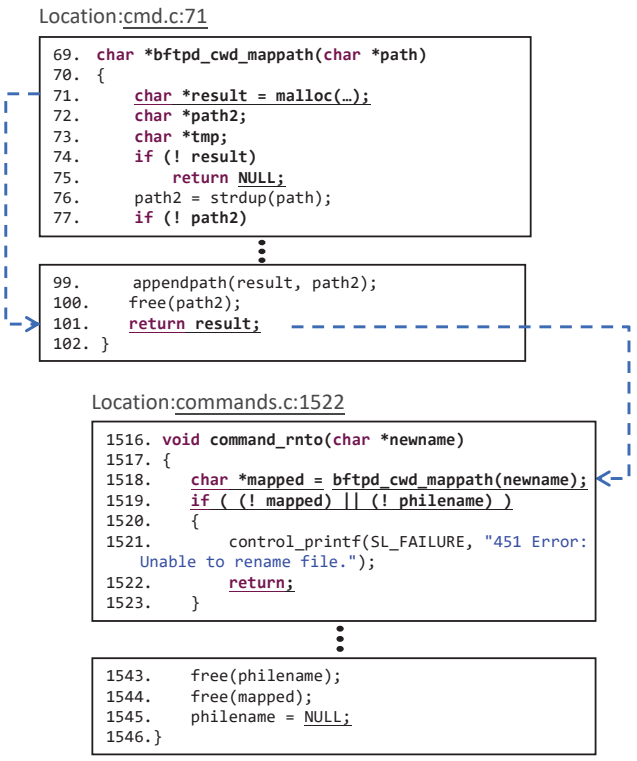


Fig. 9. A memory leak in Bftpd

In the first stage, SMOKE prunes 98.6% $(1 - 610/43,282)$ of the candidates, leaving twenty-one paths on average for each subject to be checked further. In the second stage, SMOKE further prunes 65.7% $(1 - 209/610)$ of the candidates by detecting infeasible paths. The above results indicate that both

TABLE IV
STATISTICS OF TWO ANALYSIS STAGES

Project	# Estimated Paths	1st Stage			2nd Stage		
		# Paths	Time (sec)	% Total Time	# Paths	Time (sec)	% Total Time
164.gzip	14	5	0.34	41.8%	1	0.48	58.2%
175.vpr	3	2	2.73	97.5%	2	0.07	2.5%
176.gcc	96	13	32.50	98.0%	8	0.67	2.0%
181.mcf	4	0	0.07	100.0%	0	0.00	0.0%
186.crafty	0	0	0.11	100.0%	0	0.00	0.0%
197.parser	148	3	1.82	97.6%	2	0.05	2.4%
252.eon	142	55	6.23	75.9%	4	1.98	24.1%
253.perlbmk	17	1	117.66	100.0%	1	0.02	0.0%
254.gap	1	0	0.71	100.0%	0	0.00	0.0%
255.vortex	11	5	0.97	85.1%	1	0.17	14.9%
256.bzip2	11	1	0.09	96.2%	1	0.00	3.8%
300.twolf	162	0	0.35	100.0%	0	0.00	0.0%
Bftpd	87	1	0.66	97.7%	1	0.02	2.3%
Htop	12	1	4.59	99.6%	1	0.02	0.4%
Memcached	59	3	2.50	93.7%	2	0.17	6.3%
Caffe	910	0	71.24	100.0%	0	0.00	0.0%
LAME	34	0	13.99	100.0%	0	0.00	0.0%
Zlib	1	0	0.03	100.0%	0	0.00	0.0%
Tmux	108	9	16.53	84.7%	9	2.98	15.3%
Apache httpd	10	1	11.46	100.0%	1	0.01	0.0%
OpenSSL	24	10	252.79	96.5%	6	9.14	3.5%
FFmpeg	3	3	170.61	98.0%	1	3.50	2.0%
Godot Engine	666	24	1690.00	99.5%	9	8.00	0.5%
MySQL	269	61	273.78	97.4%	17	7.22	2.6%
Chrome V8	6,152	32	2533.49	98.0%	11	50.51	2.0%
Skia	6,670	43	354.19	89.9%	11	39.81	10.1%
Blender	4,638	16	2032.35	99.9%	10	2.28	0.1%
Wine	2,011	133	259.44	98.3%	21	4.56	1.7%
Firefox	21,019	188	2377.78	99.2%	89	18.22	0.8%
Total:	43,282	610			209		
Average:			352.7	94.6%		5.2	5.4%

of the stages are effective in improving the precision.

Table IV shows the time cost of the two stages. SMOKE spends 94.6% of the time in the first phase and only 5.4% in the second phase. The two-staged design significantly reduces the cost for the path-sensitive analysis and, thus, achieves high efficiency.

F. Detected Real Memory Leaks

To better understand the usefulness of SMOKE in practice, we seek confirmations from the original developers of the subjects. Since we can not flood them with all warnings, we manually pre-screen the bug reports and choose only the ones that are likely to have severe impacts. The majority of the reports are for large and well-maintained projects such as FFmpeg, Wine, Firefox, MySQL, Godot Engine and Chrome V8. All thirty reports get confirmed by their developers and result in many patches and bug fixes. This result confirms the usefulness of our approach because those projects are regularly scanned by free and commercial tools, and SMOKE only takes $3 \sim 40$ minutes to analyze each project. We release the confirmed memory leaks online[‡].

Due to the high scalability of SMOKE, we can detect the inter-procedural memory leaks in large projects without consuming too much computing resources. Figure 10 shows

[‡]https://smokeml.github.io/list

Location: dom/media/encoder/VP8TrackEncoder.cpp:254

```
249. nsresult
250.  VP8TrackEncoder::GetEncodedPartitions(…)
251. {
252.     vpx_codec_iter_t iter = nullptr;
253.     EncodedFrame::FrameType frameType =
       EncodedFrame::VP8_P_FRAME;
254.     nsTArray<uint8_t> frameData;
```

A memory is allocated and assigned to videoData

```
277. if (!frameData.IsEmpty()) {
278.       // Copy the encoded data to aData.
279.       EncodedFrame* videoData = new EncodedFrame();
280.       videoData->SetFrameType(frameType);
281.
282.       // Convert the timestamp and duration to Usecs.
283.       CheckedInt64 timestamp = …;
284.       if (!timestamp.isValid()) {
285.           NS_ERROR("Microsecond timestamp overflow");
286.           return NS_ERROR_DOM_MEDIA_OVERFLOW_ERR;
287.       }
288.       videoData->SetTimeStamp(…);
289.
```

Take the true branch at line 284

videoData is leaked.

Fig. 10. A memory leak in Firefox

Location:SparseLU.h:445

```
429.    const Index * outerIndexPtr;
430.    if (isCompressed())
431.        outerIndexPtr = new Index[mat.cols() + 1];
432.    else
433.        outerIndexPtr = mat.outerIndexPtr();
434.
435.    for (Index i = 0; i < mat.cols(); i++) {
436.        m_mat.outerIndexPtr()[…] = outerIndexPtr[i];
437.        m_mat.innerNonZeroPtr()[…] = outerIndexPtr[i + 1]
                - outerIndexPtr[i];
438.    }
439.    if (isCompressed()){
440.        delete[] outerIndexPtr;
441.    }
```

Fig. 11. A false positive report in Blender

a memory leak confirmed by the developers of Firefox, a project of approximately eight million lines of code. In this example, the memory space pointed-to by *videoData* is allocated at Line 279 and leaks on an early return at Line 286 when *timestamp.isValid()* returns *false*. This case is not very complicated. However, because of the complexity in the code and the enormous project size, this memory leak has been hidden for more than one year. Figure 9 shows a memory leak that SMOKE found in Bftpd, a lightweight, flexible FTP server widely used in desktops, servers, embedded devices, and media centers. This vulnerability has been in the code since 2005. It has survived for more than twelve years and has impacted eighty-six versions altogether. We report this leak to its developers and receive a confirmation and an appreciation acknowledgment on its homepage.[§] A remote attacker can utilize this leak to launch denial-of-service attacks to the users of Bftpd. Due to its high severity, a CVE ID (CVE-2017-16892) is assigned.

G. Limitations

Our approach can be imprecise for several reasons. One reason is the lack of library specifications. Figure 11 shows a leak-free example simplified from a false positive report in Blender. At Line 431, the program allocates an array and stores it to the memory pointed-to by *outerIndexPtr*, under the condition that *isCompressed()* returns *true*. At Line 440, the program "deletes" *outerIndexPtr* when another call to *isCompressed()* returns *true* at Line 439. The function *isCompressed()* is implemented in a library, and it returns the same boolean value at both places. Hence, the two branches are correlated. Unfortunately, we cannot statically determine this branch correlation since we do not have the specification of the library function. One way to mitigate this problem is to manually or automatically provide specifications for libraries when detecting infeasible paths.

Another reason is that we inherit the imprecision from the lower-level program analyses such as the pointer analysis. The pointer analysis may mistakenly identify two different heap objects as aliases. Operations on one object of the alias pair may cause state changes of another, which could result in mutual interference and incorrect results.

There are still some types of infeasible paths that cannot be identified by our approach. Such paths usually involve complex arithmetic in branch conditions, complicate data dependence or deep inter-procedural effects. Including more control and data dependence in the path constraints can help to mitigate this limitation, at the price of being less efficient.

VI. Related Work

In this section, we survey the related approaches in two categories.

A. Static Memory Leak Detection

SATURN [3] reduces the memory leak detection problem to a boolean satisfiability problem and uses a SAT-solver to find memory leaks. Facebook INFER [2] is based on bi-abductive inference of separation logic that extends the Hoare Logic by explicitly modeling the heap. CLOUSEAU [4], [5] detects memory leaks based on a practical ownership model of memory management. Orlovich and Rugina [6] proposed a leak detection algorithm based on a reverse data-flow analysis that assumes the presence of the leak first, followed by a pruning method. SPARROW [7] detects memory leaks in C programs based on abstract interpretation. Tools based on symbolic execution such as the CSA [1], COVERITY SAVE [28], and KLEE [29] are promising in practice. By treating all external inputs as symbols and execute the program on symbols, symbolic execution tools can explore program states that are hard to reach for concrete executions. However, symbolic execution tools do not scale to large programs due to the path explosion problem and intensive uses of the constraint solver. All of the above techniques are not sparse, which analyze a lot of irrelevant program statements and, thus, suffer from performance issues.

FASTCHECK [8], SABER [9], and PINPOINT [18] are mostly related to our approach. They work on a sparse value flow graph with guards annotated on the graph edges. As we have discussed in this paper, when high precision is required, VFG-based model of the memory leak problem [8], [9] is the reason for the analysis to be not scalable. To achieve scalability, FASTCHECK and SABER have to compromise the path-sensitivity, leading to many false warnings. PINPOINT improves the sparse value-flow analysis by building precise data dependence in an efficient manner, but it does not provide a better model for memory leak detection. In comparison, we propose a FSM-based model that enables us to efficiently detect memory leaks on UFG.

B. Dynamic Memory Leak Detection

Dynamic approaches [24], [30]–[36] detect memory leaks by instrumenting and running a program. Some dynamic approaches operate at the binary level, such as the memcheck tool of VALGRIND [31], DR. MEMORY [30] and PURIFY [33]. These approaches track memory allocation and deallocation during a program's execution, and detect leaks by scanning the program's heap for memory blocks that no pointer points to. INSURE++ [36] and ADDRESSSANITIZER [35] detect memory leaks by inserting extra statements to the source code before compiling the binary.

Unlike static approaches that have relatively high false positive rate due to the abstraction of concrete program states, dynamic approaches have few false positives because they have access to the concrete program states at runtime. However, dynamic approaches can miss many real bugs because they cannot cover all possible program behaviors with limited number of test cases. Also, dynamic approaches are hard to be applied in a production run, because running the instrumented program usually causes unbearable runtime overhead.

VII. Conclusion

We have presented an approach to static memory leak detection, which runs in a fast, scalable, and precise manner. The key factor to make our technique fast is a staged analysis, in which we first efficiently filter safe cases based on our new program representation, i.e., use-flow graph, and then employ a constraint solver to verify path feasibility only for a handful of leak candidates. We implemented our technique as a tool, called SMOKE, and evaluated it systematically. The evaluation results demonstrate that SMOKE is promising as an industrial-strength static memory-leak detector.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and Dr. Yulei Sui for his help on the SABER implementation. This work was partially funded by Hong Kong GRF16214515, GRF16230716, GRF16206517, NSFC61628205 and ITS/215/16FP grants.

[§] http://bftpd.sourceforge.net/news.html

REFERENCES

[1] The LLVM Foundation, "Clang static analyzer," https://clang-analyzer.llvm.org/, 2018.

[2] Facebook, Inc., "Infer," 2018. [Online]. Available: http://fbinfer.com/

[3] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 115–125.

[4] D. L. Heine and M. S. Lam, "Static detection of leaks in polymorphic containers," *Proceeding of the 28th international conference on Software engineering - ICSE '06*, p. 252, 2006.

[5] ——, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," *ACM SIGPLAN Notices*, vol. 38, no. 5, p. 168, 2003.

[6] M. Orlovich and R. Rugina, "Memory Leak Analysis by Contradiction," *Proceedings of International Static Analysis Symposium SAS06*, 2006.

[7] Y. Jung, "Practical Memory Leak Detector Based on Parameterized Procedural Summaries," in *Proceedings of the 7th International Symposium on Memory Management*, 2008.

[8] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 480–491. [Online]. Available: http://doi.acm.org.lib.ezproxy.ust.hk/10.1145/1250734.1250789

[9] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, Feb 2014.

[10] Mozilla, "Mozilla bugzilla," 2018. [Online]. Available: https://bugzilla.mozilla.org/buglist.cgi?quicksearch=memory+leak

[11] Google, "Chromium bugs," 2018. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/list?can=1&q=memory+leak

[12] "CVE List," 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=memory+leak

[13] S. Zacchiroli, "The debsources dataset: Two decades of debian source code metadata," in *IEEE International Working Conference on Mining Software Repositories*, 2015.

[14] S. McPeak, C.-H. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, 2013.

[15] A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, and S. McPeak, "A few billion lines of code later," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb 2010.

[16] B. Steffen, B. Steffen, J. Knoop, and O. Rüthing, "The value flow graph: A program representation for optimal program transformations," *PROCEEDINGS OF THE EUROPEAN SYMPOSIUM ON PROGRAMMING, PAGES 389–405. SPRINGER-VERLAG LNCS 432*, vol. 432, pp. 389–405, 1990.

[17] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158.

[18] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the ACM SIGPLAN 2018 Conference on Programming Language Design and Implementation*, ser. PLDI '18. Philadelphia, PA, USA: ACM, 2018.

[19] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning, "A deterministic (2-2/(k+1))n algorithm for k-sat based on local search," *Theoretical Computer Science*, vol. 289, no. 1, pp. 69 – 83, 2002.

[20] N. Bjorner and L. de Moura, "Z3: An efficient SMT solver," *Available from:http://research.microsoft.com/projects/Z3*, 2007.

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," 1991.

[22] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61.

[23] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[24] C. Jung, S. Lee, E. Raman, and S. Pande, "Automated memory leak detection for production use," *Proceedings of the 36th International Conference on Software Engineering*, no. undefined, pp. 825–836, 2014.

[25] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *ECOOP'95*, 1995.

[26] R. DeLine and M. Fähndrich, *Typestates for Objects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 465–490.

[27] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *Computer*, 2000.

[28] "Coverity scan," https://scan.coverity.com/projects/, 2018.

[29] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[30] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 213–223.

[31] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," 2007.

[32] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 515–524.

[33] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," in *Proceedings of the Usenix Winter 1992 Technical Conference*, 1991.

[34] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," *ACM SIGOPS Operating Systems Review*, 2004.

[35] K. Serebryany and D. Bruening, "AddressSanitizer: a fast address sanity checker," *ATC*, 2012.

[36] "Insure++," 2018. [Online]. Available: http://www.parasoft.com/products/insure