

# Efficient and linear static approach for finding the memory leak in C

Vishruti Desai<sup>1,2</sup>, Vivaksha Jariwala<sup>3,4</sup>

<sup>1</sup>Ph.D Section, Gujarat Technological University, Ahmedabad, India

<sup>2</sup>Computer Engineering Department, C K Pithawala College of Engineering and Technology, Surat, India

<sup>3</sup>Ph.D Section, Gujarat Technological University, Ahmedabad, India

<sup>4</sup>Sarvajanik College of Engineering and Technology, Sarvajanik University, Surat, India

## Article Info

### Article history:

Received Mar 3, 2022

Revised Oct 7, 2022

Accepted Nov 1, 2022

### Keywords:

Embedded system

Memory leak

Sparse value flow

Static analysis

## ABSTRACT

Code analysis has discovered that memory leaks are common in the C programming language. In the literature, there exist various approaches for statically analyzing and detecting memory leaks. The complexity and diversity of memory leaks make it difficult to find an approach that is both effective and simple. In embedded systems, costly resources like memory become limited as the system's size diminishes. As a result, memory must be handled effectively and efficiently too. To obtain precise analysis, we propose a novel approach that works in a phase-wise manner. Instead of examining all possible paths for finding memory leaks, we use a program slicing to check for a potential memory leak. We introduce a source-sink flow graph (SSFG) based on source-sink properties of memory allocation-deallocation within the C code. To achieve simplicity in analysis, we also reduce the complexity of analysis in linear time. In addition, we utilize a constraint solver to improve the effectiveness of our approach. To evaluate the approach, we perform manual scanning on various test cases: link list applications, Juliet test cases, and common vulnerabilities and exposures found in 2021. The results show the efficiency of the proposed approach by preparing the SSFG with linear complexity.

*This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.*



## Corresponding Author:

Vishruti Desai

Ph.D Section, Gujarat Technological University

Chandkheda, Ahmedabad, Gujarat

Email: Vishruti.phd@gmail.com

## 1. INTRODUCTION

Long-running embedded systems are susceptible to memory leaks. However, if the attacker can deliberately cause a memory leak, they may be able to carry out a denial-of-service attack (by crashing the application) or exploit other unusual program behavior brought on by insufficient memory. Generic software reliability issues are generally brought forward by memory leaks [1], [2]. When software acquires a block of memory but does not release it after its last usage, it is called a memory leak. If one or more accessible pointers still refer to such a block after the execution, repairing the leak is usually straightforward as long as the location of the block is known [3], [4]. Nevertheless, if all references to the block are changed or removed while the program is running, knowing the allocation site is often insufficient. In any event, a memory leak occurs when a developer forgets to remove a memory allocation when it is no longer needed, resulting in accidental memory consumption [5], [6]. Finding memory leaks properly and with less complexity is still a difficult task, despite the considerable research improvements made recently. To date [7], 1,219 records for memory leaks in common vulnerabilities and exposures (CVE) have been found. The efficiency and

complexity criteria of analyzing code to detect memory leaks must be addressed as the complexity of the code increases [8]–[10].

In recent years, various approaches such as taint analysis (e.g., leak point [11]), sparse value flow analysis (e.g., fast check [12]), data flow analysis (e.g. contradiction [13], PCA [14]), Boolean satisfiability (e.g. Saturn [15]), abstract interpretation (e.g. sparrow [16]), path-insensitive flow analysis (e.g. saber [5]), type-specific (e.g. leak fix [7]), value flow graph (e.g. smoke [17]), ownership reasoning (e.g. k-meld [18]) are proposed for static analysis methods as detecting memory leak, whereas dynamic analysis is performed when code is in execution. Saha *et al.* [19] have created a prototype Hector based on the resource-release concept that is designed for the system software's error handling module. They have employed specification mining, but only for intra-procedure. Even Machiry *et al.* [20] developed Dr. Checker that worked on static analysis specifically for kernel driver programs. They have considered point-to-analysis and taint analysis. Although both their work and our work are closely related, we also choose our own representation of the program code to interpretation for the program's complexity. This makes our strategy simple and efficient. We observed that the complex approach lost its precision whereas the precise approach limits complexity. Hence, our focus is on program representations. Table 1 shows the various tools along with attributes like complexity (C), their internal program representations, and precision (P).

Table 1. Comparison of various memory leak detectors

Sr. No.	Tool	Program Representations	Platform	P	C	Remarks	Citation
1.	Fastcheck	Guarded value flow analysis	Crystal	N	N	Partial sparse value flow (SVF), heavy weighted	[12]
2.	Leak point	Control flow graph	LLVM	N	Y	Perform taint analysis for allocation	[11]
3.	Saber	Full sparse value flow graph	Open64	Y	Y	Full representation of code as SVF	[5]
4.	Leak fix	Control flow graph	LLVM	N	Y	Finding leak and insert multiple fixes too	[3]
5.	Smoke	User data flow	LLVM	N	Y	Heavy weight	[17]

From Table 1, we also observed that the control flow graph (CFG) and SVF were two variants as program representations. We took into account both program representations. We discovered that a control flow graph is widely used in literature to demonstrate a program's control and that the SVF is also appropriate for pointer analysis. In such a situation, we combine the applicability of both representations to increase the effectiveness and precision of our approach. The source-sink property [5] is largely used to address the memory leak problem, which is also characterized as a finite state machine problem [21]. The SVF, which was created to improve code analysis, does not fit well with the characteristics of randomly generated finite state machines. We reconstruct the SVF to have a source-sink flow graph (SSFG) based on the source-sink properties in the context of this approach. The main objective of our approach is to follow the allocation pointer throughout the control path from source to sink. Its underlying SSFG analysis must be efficient and effective.

In this paper, we focus on important challenges including memory leaks for embedded systems' memory usage [22]. The software suffers because of it. Memory leaks typically occur when a developer forgets to release memory before terminating an application. If memory leaks happen repeatedly, memory loss will eventually develop and affect the system's overall reliability and performance [3]. According to the points below, the memory leak is significant in a variety of ways.

- Developing system software and embedded device software is best done in a language like C. In contrast to managed programming languages such as Java, C requires external memory management to keep track of the memory leaks.
- It supports future programs in addition to the one that is now in service. All programs executing on the system are impacted by a program that leaks memory in terms of efficiency and durability.
- Before the system has worn down, it is quite difficult to notice.
- Generic bug-fixing approaches, such as asserting statements like "no leak," are ineffective.
- Additionally, it predominates in more sophisticated applications.

The objective of this paper is to present a simple method for C source code that effectively and with linear complexity discovers memory leaks. Figure 1 illustrates the efficiency of our approach and provide an effective analysis of the source code. Precision and effectiveness are key concepts in our consideration of complexity reduction to find memory leaks.

Traditional sparse value flow analysis (SVFA) methods employ partial or fully sparse value flows. [2]–[4]. Figure 1(b) displays the SVF graph representation for a constraint condition to find the leak for a program part Figure 1(a). Our notation denotes the memory pointer  $p$  at statement  $st$ , as illustrated in node. The data dependency relationship between the statements is indicated by the data value flow on the graph edges. The constraint solver is employed to address the particular condition and determine if memory is

released correctly or not. If the function  $F(no\ leak)=c1 \vee (! c1 \wedge c2)$  meets the requirements, there is a path from the allocation site to the free site in the segment, proving that the susceptibility to memory leaks does not exist. With  $k \geq 3$  and exponential time complexity, K-SAT is currently the constraint solver that is employed in the literature. In our approach, we built a novel way of sparse program representation called the Source-Sink Flow Graph (SSFG). In SSFG, the data value flow that is present in SVFA is still around. Additionally, the allocation of objects from the allocation site to the free sites is tracked using CFG. In our SSFG, we impartially include the statements needed for control tracing and remove the portions that are irrelevant. The SSFG form for the same code segment is presented in Figure 1(c). When the program terminates, a simple path traverse in SSFG reveals that the pointer  $p$  is never released, while the same path in SVF cannot be found. In that sense, the complexity of our representation will be linear as compared to exponential. Also, for a variety of reasons, traditional approaches also integrate annotation (i.e.,  $\mu$ ) and value flow path. It raises the codebase overhead of a large application. In contrast, we took into account path slicing in our approach to increase precision while decreasing the total time and space costs for path validation [23].

Based on source-sink characteristics of memory utilization, we develop a modified source-sink flow graph (SSFG) of program representation. We propose a comprehensive static method that is effective and simple for identifying memory leaks. We examine the effectiveness of our novel approach using a variety of case study applications, test suits, and authentic CVE.

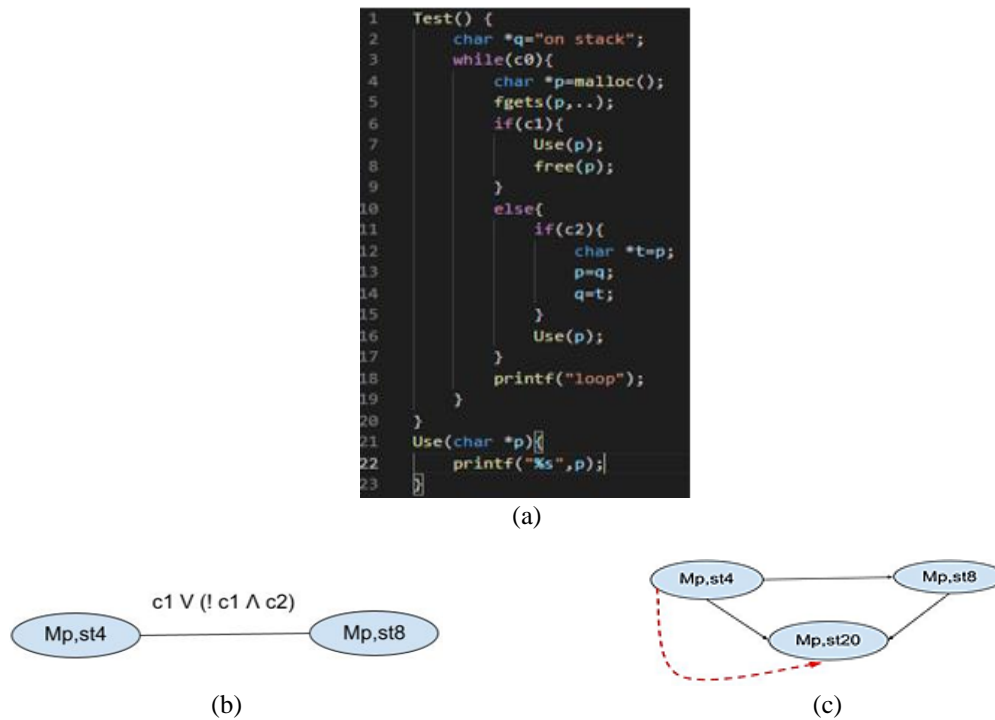


Figure 1. Memory leak program in C (a) source code in C program, (b) value flow graph, and (c) source-sink flow graph

## 2. PROPOSED APPROACH

Our approach consists of a four-phase examination of the code. The phase-wise flow of our proposed approach is shown in Figure 2. We have taken into consideration CFG and point-to-analysis data in the first phase. Using CFG, we get information on the control flow of the static execution of particular input. Additionally, the point-to-analysis method enables flow-insensitive analysis. We have used Anderson's point [24] to analyze to prepare the set of relevant points needed for our purpose. It is an alias analysis that is flow-insensitive, context-insensitive, and field-insensitive. That is used for inter-procedure interaction. It gathers the point to set for all pointer variables on user inputs. As a consequence, it offers a high level of precision, which is commonly used in modern compilers and business scenarios. In order to build the call graph and control flow graph in the second phase, we use the implicit or explicit callee and caller information from point-to-sets gained in the first phase. This particular result is obtained for inter-procedure analysis

utilizing the gold plug (i.e. tool used for cross compilation) and low level virtual machine (LLVM) [25] call graph. Furthermore, we also obtain the pointer variable's precise calculation for memory allocation.

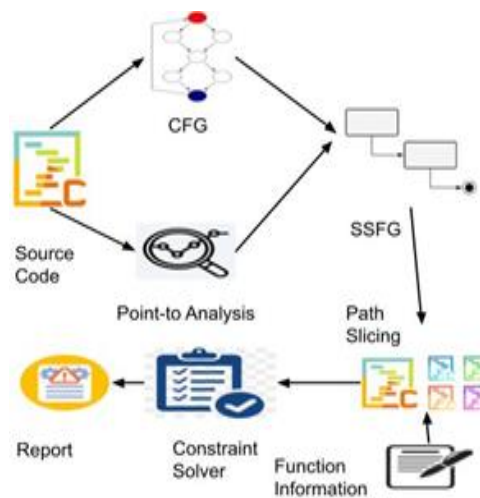


Figure 2. Proposed approach

In the next phase, we illustrate a source-sink flow graph (SSFG) that uses the data acquired in the preceding phase to track the allocation and free sites of particular variables. We use reconstructed lightweight state machine characteristics for SSFG. By using this, we can identify memory leaks more efficiently and execute our code at a faster rate. Generation of SSFG is the core part of our approach, which we go through in further detail later on. In the subsequent phase, Path Slicing is used to extract unnecessary statements from the code while keeping its objective. Backtracking across the graph from nodes that meet the slicing conditions will provide the collection of instructions that compose a slice. We take into account both static slicing, which runs on all potential input, and backward slicing, which locates the problematic part of the program. We employ context-sensitive, lightweight symbolic program slicing (SymPas) [26] that takes into consideration both slicing methods. Along with path slicing, we also add procedure summaries for specific modules to be more detailed and context-sensitive [16] information. In the last phase, we applied path slicing and a constraint solver to enhance the precision of our approach [27]. Algorithm 1 presents the phase-wise steps of the proposed approach.

#### Algorithm 1. Algorithm to locate memory leak

Input: Source code written in C

Output: Warning of memory leak that contains location and variable name

ASSUMPTION: Program is in Static Single Assignment (SSA) Form

1. Build CFG and Perform Point-to Analysis
2. Build Source-Sink Flow Graph
3. Perform Path Slicing using Function Summary Information
4. Apply Constraint Solver on Slice path
5. Generate Report

Since every object is created at the allocation site (source) and deallocated site (sink). Allocation-deallocation is mapped as a source-sink (SS) property for discovering memory leaks in C programs. We define a source-sink flow graph (SSFG) to make use of this property and the state transition graph (finite automata) characteristics. In terms of state transition characteristics, the SSFG offers benefits such as i) it keeps track of a control flow graph that depicts the whole life cycle of the variables and is represented as a machine transition flow, ii) regarding program size, it creates transition flow in a linear amount of time, and iii) it verifies the state machine flow and performs the required steps in the correct sequence.

According to Definition 1 in the literature, memory instructions (allocate and free) in source code are used to define state transitions [28]. The transition diagram is formally represented in Definition 1. Definition 1: A finite automation is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where  $Q$  is a set of memory program statements (st) (i.e., allocate, deallocate),  $\Sigma$  is a set of variables in memory statements,  $\delta: Q \times \Sigma \rightarrow Q$  is the transition function, that is  $\delta(m_1, st_1) \rightarrow st_2$ ,  $q_0 \in Q$  is the start (or initial) state, and  $F \subseteq Q$  is the set of accepted (or final) states.

Given a transition diagram in Figure 3 and Definition 1, construct SSFG as follows for the variable life cycle. We use this Definition 1 to generate the specification for our SSFG presentation in Definition 2. Definition 2: Given a program source-sink flow graph (SSFG) is a directed graph  $G=(V, E)$  where  $V=\{\text{set of statements: ALLOCATE, DEALLOCATE, INVALID, END}\}$  and  $E=\{\text{set of directions: FREE, NO_SCOPE, MALLOC}\}$ .

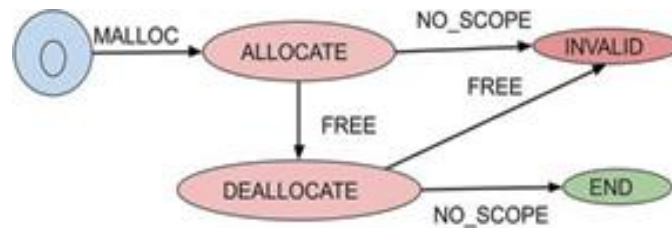


Figure 3. Transition diagram

Consider A vertex  $(M_p, St) \in V$  in which the memory variable  $M_p$  at one memory program statement  $St$  and is any of the following: i)  $St \in \Sigma$ , means  $st$  is a program statement in that it takes to transition from one state to another; ii)  $St$  is either the actual parameter  $M_p$  at caller or formal parameter at callee  $M_p$ ; and iii) lastly, for every  $(M, St')$  is dominance frontier of  $St$ . For Edge,  $((M_p, St), (M_p, St'))$  if and only if there is a control path from  $St$  to  $St'$ .

For every section of program code, as described in Definition 2, we developed SSFG. Our main impartial is to examine the CFG for each function and eliminate any blocks of instructions that are obviously unnecessary. As a result, we construct a transition table to analyze variables affected by memory statements to check if a memory leak exists. We take into account the two distinct analyses presented below to depict the construction of SSFG and find memory leaks in them.

### 2.1. Code 1: intra procedure analysis

Figure 4(a) illustrates the working steps of our program's analysis. First, as illustrated in Figure 4(b), we construct the CFG. Each node of CFG contains the fundamental building blocks of the LLVM intermediate representation (IR) form for each function. We simplified this representation as shown in Figure 4(c) using our Definition 2 notations. For each memory variable contained in the specified source code Figure 4(a), a transition table was built by taking into account Figures 3 and 4(c). Figure 4 (d) shows each entry in the state table that follows the control path for a specific variable. Variable  $p$  has allocated memory at line 3 in our sample program. The dominance frontier (DF), a node where various pathways from the same variables are merged, has also been taken into consideration. The DF is traversed by each variable. Therefore, the CFG gives us the relevant DF node for the analysis. In our scenario, step 4 of the state table is the dominating frontier because it merges two distinct variable states. In step 4, we see that the states of variable  $p$  transit from allocate (A) to invalid (I), signifying an error, and variable  $q$  transit from deallocate (DA) to end (E), signifying the definitive result of the sequence. Our finite automata predicts that a portion of code suffers a memory leak whenever a state transits from A to I without a scope. The path slicing and constraint solver of the approach will now verify this path.

### 2.2. Code 2: inter procedure analysis

Examine the inter-program in Figure 5(a) in detail for program representations. A specific CFG and call graph for the program as produced in the approach's first phase are also shown in Figures 5(b) to 5(d). We presented the SSFG and state table for the program in Figure 5(a) to illustrate how we would proceed in the next phases.

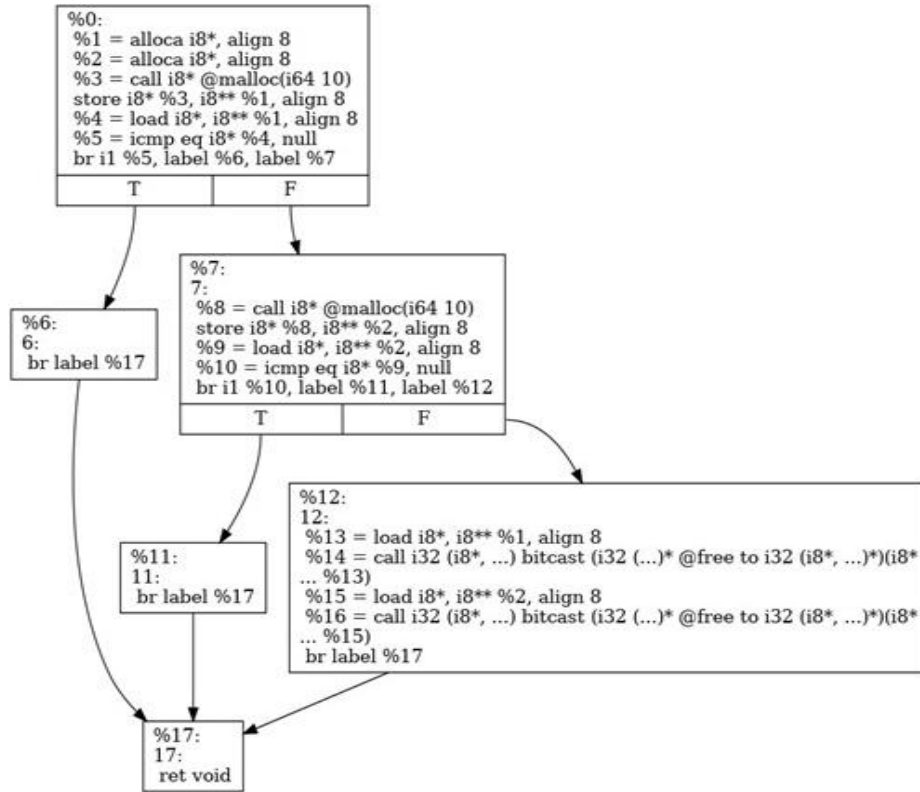
In next phase, we perform program representation in our notations of SSFG form. Here, Figures 6(a) and 6(b) shows both components of our approach for building the SSFG. Additionally, in step 7 of the state table, two paths are merged. Line 8 of the program in this example represents the DF. In this instance, a no-scope operation is used to acquire deallocate (DA) to end (E), which is seen as a memory leak. Path slicing again supported SSFG's claim of a memory leak in this instance.

Additionally, we found that studies consider the CFG or SVF when analyzing whether or not the program has memory leaks. However, in our scenario, we took into account both the methods and the DF condition node to determine if the current path leads to a memory leak or not. We developed the approach for that using both inter- and intra-procedural scenarios.

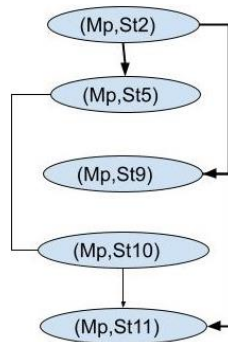
```

1 void ML() {
2     int *p = malloc();
3     if (p == NULL)
4         return -1;
5     int *q = malloc();
6     if (q == NULL)
7         return -1;
8     ...
9     free(p);
10    free(q);
11 }
    
```

(a)



(b)



(c)

Step	State	Input	Output
1	(Mp,St2)	-	PA
2	(Mq,St5)	PA	PA, QA
3	(Mq,St10)	PA, QA	PA, QDA
4	(Mpq,St11)	PA, QF	PI, QE

(d)

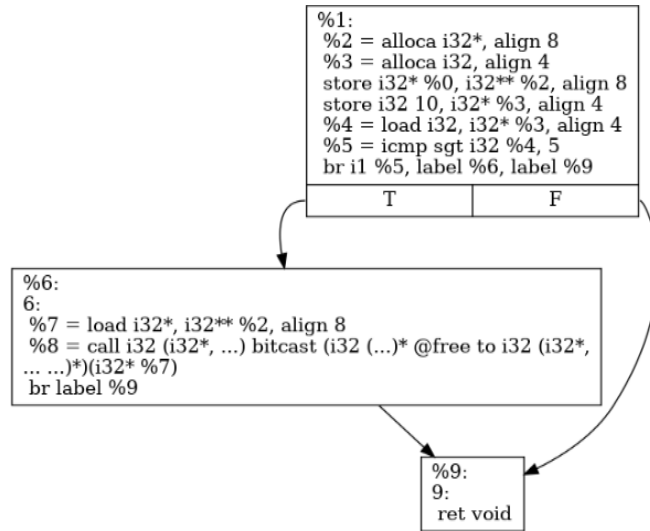
Figure 4. Intra procedure analysis; (a) program with memory leak, (b) CFG in LLVM IR form, (c) SSFG, and (d) state table



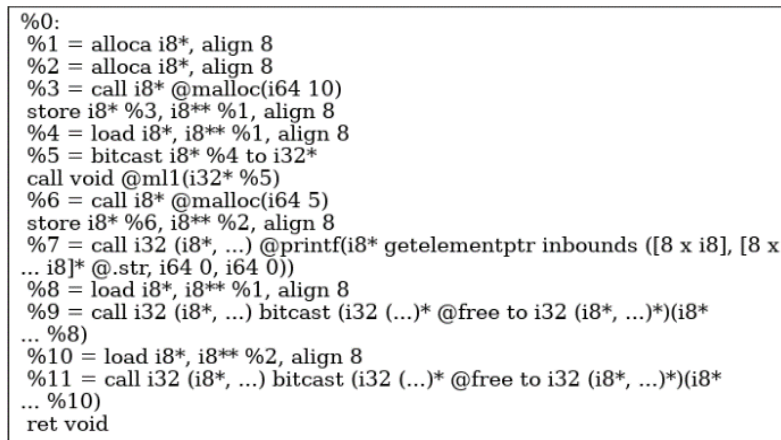
```

1 void ml()
2 {
3     char *p=malloc(10);
4     m1(p);
5     ...
6     if(...)
7         free(p);
8 }
9 void m1(int *q){
10     if(...)
11         free(q);
12 }
    
```

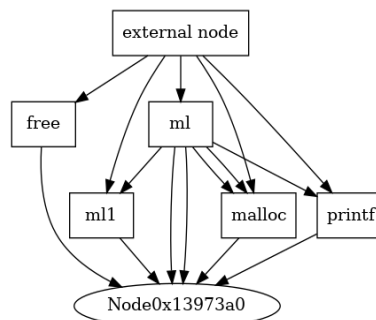
(a)



(b)



(c)



(d)

Figure 5. Inter procedure source code with CFG and call graph; (a) inter procedure program, (b) CFG for ‘ml1’, (c) CFG for ‘ml’, and (d) call graph

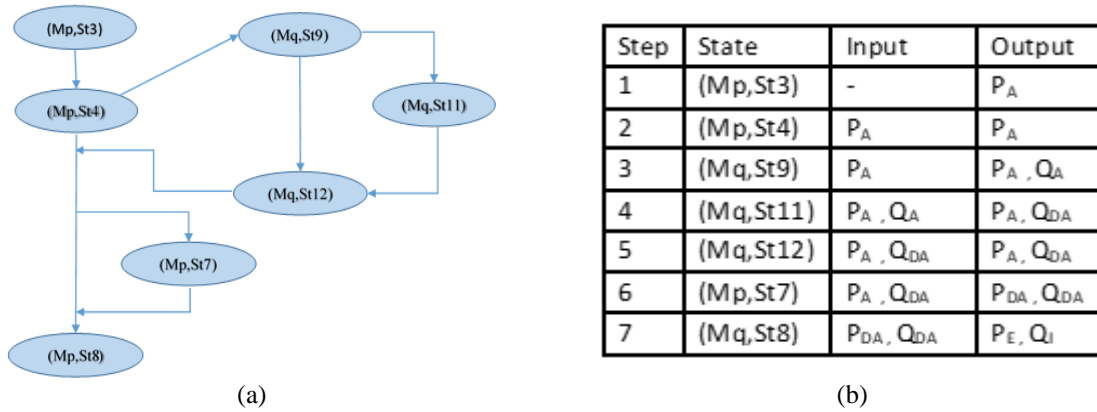


Figure 6. Inter procedure SSFG: (a) CFG in synthesized form and (b) state table

### 3. RESULTS AND DISCUSSION

In that instance, we assume that the source code is in the static single assignment (SSA) format. It is an intermediate representation (IR) that analysis can quickly handle. Each source program that is intended for the given application is given an IR form by the LLVM gold plugin. Additionally, a developer can analyze a specific function or fragment of code to raise the effectiveness of the approach. In a variety of scenarios, we evaluated the effectiveness of functional representations and their application to the analysis of memory leaks over time. In this case, we considered various scenarios developed for the verification of approaches. We tested our proposed approach using various linked list (LL) applications and a legitimate memory leak identified by CVE. We also took into account the Juliet Test suit V3.0 CWE401 [29] test scenarios. It has a number of instances of memory leaks.

We manually validated a number of test scenarios, which might increase the likelihood of false positives. We cannot support dynamically loaded files, not even for source code, due to our constraints. We first examine the effectiveness of our approach for detecting memory leaks using LL test applications. We manually created memory leaks in the LL applications for that. Secondly, Privoxy's handling errors module was found to have a memory leak flaw (CVE-2021-44542). We developed CFG and contrasted it with CFG produced by LLVM. We thoroughly retraced the program and created SSFG in accordance with Definition 2. Our pre-scan of the SSFG path for specific CVEs revealed that our approach is successful at locating the memory leak in Privoxy's out-of-date code. To make sure that our approach is efficient and linear as well, we extensively validated them using the contexts of our respective SSFG and constraint solvers. Figure 7 depicts the real breach that is found in CVE Privoxy. In the program, to\_send was initially constructed by the developer at line number 2187 in the earlier code, but it was never released. The memory leak is fixed by inserting the deallocation statement at line number 2209. By carefully tracing the whole older source code, we were able to confirm that the memory leak was the same.

```

2183 static int send_http_request (struct client_state *csp)
2184 {
2185 char *hdr;
2186 int write_failure;
2187 const char *to_send;
2188 size_t to_send_len;
2189 int filter_client_body = csp->expected_client_content_length != 0 &&
2190 client_body_filters_enabled(csp->action) && can_filter_request_body(csp); 219
...
...
2199 to_send = execute_client_body_filters (csp, &to_send_len);
2200 if (to_send == NULL)
2201 {
2202 /* just flush client_iob */
2203 filter_client_body = FALSE;
2204 }
2205 else if (to_send_len != csp expected_client_content_length && 2206 update_client_headers
(csp, to_send_len))
2207 {
2208 log_error (LOG_LEVEL_HEADER, "Error updating client headers");
2209 freez(to_send);
2210 return 1;

```

Figure 7. Memory leak in Privoxy



Third, we have considered the memory leak contain in Juliet test Suits. Figure 8 shows a sample test case from Juliet CWE401 specifically for a memory leak. By constructing phases of our approach and manually tracing programs, we validate these test cases as well. In all these cases, we have not considered the API and global variables available for the whole code base.

```

1 void CWE401_Memory_Leak_char_calloc_01_bad()
2 {
3     char * data;
4     data = NULL;
5     data = (char *)calloc(100, sizeof(char));
6     if (data == NULL) {exit(-1);}
7     strcpy(data, "A String");
8     printLine(data);
9 }
10 int main(int argc, char * argv[])
11 {
12     ...
13     CWE401_Memory_Leak_char_calloc_01_bad();
14     ...
15 }

```

Figure 8. CWE401 sample program

We analyzed all three test scenarios manually and looked at the following research questions to measure the effectiveness and efficiency of our approach.

RQ1: How effective is our proposed approach in finding memory leaks?

RQ2: How can memory leaks be located as precisely and with less complexity as possible?

### 3.1. Effectiveness and efficiency

To gather the information required to study RQ1, we utilized the four separate programs. In our approach, each phase was developed as a separate module that was performed and independently tested for input and output. For efficiency, we focused on the following standalone programs, test suitcases, and verified CVE memory leaks. Table 2 shows the findings of our investigation as well as the number of leaks that were found. We observed that high proportion of precision for basic programs like LL, but at the same time precision ratio drops when we investigate large-scale programs for verification. Due to the program's usage of dynamic files, which we did not take into account while evaluating it, the effectiveness percentage reduces. As a result, even though further study is required, we believe our findings to be positive.

Table 2. Analysis of memory leak

Application	Size of Code (KLOC)	# Functions / Test Cases	Our Approach		Precision %
			Present	Found	
Program 1	150	6	3	3	100 %
Program 2	250	10	6	5	83.34%
Juliet Test Suit V3.0	-	2826	531	389	73.26%
CVE-2021-44542	6K	255	2	1	50%

### 3.2. Precision and complexity

We phased our approach to doing thorough research for RQ2. In our opinion, the two phases cross-verify the same path when our new notation of SSFG identifies a memory leak along it, increasing the approach's precision. Because we carried out each phase independently on each module, we could have lost some path verification and program correctness. We lost some program memory since we utilized point-to as a baseline, which was not constructed for alias values. We take finite automata with an  $O(n)$  level of complexity, where  $n$  is the number of inputs that reduce the complexity of the approach. The approach has linear complexity as a result. Precision is provided via path matching with inputs. It claims that finding a memory leak in programs is made simple and efficient using the approach we proposed.

## 4. CONCLUSION

We developed a novel memory leak detection approach to program representations in this paper. It locates leaks across an execution in addition to detecting them. The path to take a particular variable from source to sink is determined by the SSFG phase. We propose a phase-wise approach where we first identify a

memory leak in a software application and then verify it using the next two phases. The validity of our suggested approach is probably being challenged by the existing constrained analysis of it based on a small number of applications. In order to provide a reference set of practical applications to evaluate memory leak detection algorithms, we look at open-source applications that have already encountered memory leaks. Our approach has linear complexity, we also reduced its precision for the tested application of our result. In the future, we want to undertake our research using this group of applications, as well as brand-new ones with seeded memory flaws, alongside our industrial partner and college students.





## REFERENCES

- [1] L. Yuan, S. Zhou, and N. Xiong, "MLD: An intelligent memory leak detection scheme based on defect modes in smart grids," *arXiv preprint arXiv:2008.09758*, pp. 1–13, Aug. 2020.
- [2] M. M. Joy, W. Mueller, and F. Rammig, "Early phase memory leak detection in embedded software designs with virtual memory management model," in *Proceedings of the 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop December 3 Vancouver Canada*, Nov. 2013, vol. 90, pp. 25–28. doi: 10.3384/ecp13090005.
- [3] H. Yan, Y. Sui, S. Chen, and J. Xue, "Automated memory leak fixing on value-flow slices for C programs," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, Apr. 2016, pp. 1386–1393. doi: 10.1145/2851613.2851773.
- [4] V. Šor, S. N. Srirama, and N. Salnikov-Tarnovski, "Memory leak detection in Plumb," *Software - Practice and Experience*, vol. 45, no. 10, pp. 1307–1330, 2015, doi: 10.1002/spe.2275.
- [5] 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, doi: 10.1109/TSE.2014.2302311.
- [6] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSA 2012*, 2012, pp. 254–264. doi: 10.1145/2338965.2336784.
- [7] CVE, "CVE-2020," CVE Records, 2022. Accessed: Mar. 01, 2022. [Online]. Available: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=memory+leak>
- [8] J. Wang, X.-D. Ma, W. Dong, H.-F. Xu, and W.-W. Liu, "Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis," *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 347–356, Mar. 2009, doi: 10.1007/s11390-009-9229-0.
- [9] B. Yu, C. Tian, N. Zhang, Z. Duan, and H. Du, "A dynamic approach to detecting, eliminating and fixing memory leaks," *Journal of Combinatorial Optimization*, vol. 42, no. 3, pp. 409–426, Oct. 2021, doi: 10.1007/s10878-019-00398-x.
- [10] Z. Z. Xu *et al.*, "Symbolic pointer analysis for detecting memory leaks," *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, vol. 2, no. 1, pp. 104–112, Feb. 2007, doi: 10.1145/328691.328704.
- [11] J. Clause and A. Orso, "LEAKPOINT: Pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 2010, vol. 1, p. 515. doi: 10.1145/1806799.1806874.
- [12] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 480–491, Jun. 2007, doi: 10.1145/1273442.1250789.
- [13] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2006, pp. 405–424. doi: 10.1007/11823230\_26.
- [14] W. Li, H. Cai, Y. Sui, and D. Manz, "PCA: memory leak detection using partial call-path analysis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2020, pp. 1621–1625. doi: 10.1145/3368089.3417923.
- [15] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 115–125, Sep. 2005, doi: 10.1145/1095430.1081728.
- [16] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proceedings of the 7th international symposium on Memory management - ISMM '08*, 2008, p. 131. doi: 10.1145/1375634.1375653.
- [17] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "SMOKE: Scalable path-sensitive memory leak detection for millions of lines of code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, vol. 2019-May, pp. 72–82, doi: 10.1109/ICSE.2019.00025.
- [18] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *Proceedings 2021 Network and Distributed System Security Symposium*, 2021, no. February, doi: 10.14722/ndss.2021.24416.
- [19] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2013, pp. 1–12, doi: 10.1109/DSN.2013.6575307.
- [20] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. Checker: A soundy analysis for linux kernel drivers," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 1007–1024.
- [21] Z. Xu, J. Zhang, and Z. Xu, "Memory leak detection based on memory state transition graph," in *2011 18th Asia-Pacific Software Engineering Conference*, Dec. 2011, pp. 33–40. doi: 10.1109/APSEC.2011.22.
- [22] Y. Dong, W. Yin, S. Wang, L. Zhang, and L. Sun, "Memory leak detection in IoT program based on an abstract memory model SeqMM," *IEEE Access*, vol. 7, pp. 158904–158916, 2019, doi: 10.1109/ACCESS.2019.2951168.
- [23] Q. Gao *et al.*, "Safe memory-leak fixing for C programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015, vol. 1, no. 1, pp. 459–470. doi: 10.1109/ICSE.2015.64.
- [24] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, Feb. 2016, pp. 160–170. doi: 10.1145/2854038.2854043.
- [25] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. doi: 10.1109/CGO.2004.1281665.
- [26] Y.-Z. Zhang, "SymPas: Symbolic program slicing," *Journal of Computer Science and Technology*, vol. 36, no. 2, pp. 397–418, Apr. 2021, doi: 10.1007/s11390-020-9754-4.
- [27] Z. Rakamarić and M. Emmi, "SMACK: Decoupling source language details from verifier implementations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8559 LNCS, no. Iv1, 2014, pp. 106–113. doi: 10.1007/978-3-319-08867-9\_7.





- [28] J. Hopcroft, J. E., and Ulman, *Introduction to automata theory, languages, and computation*, 2nd ed. Addison-Wesley, 2001.
- [29] NIST, "Juliet Test Suit V1.3 for C," NIST. Accessed: Feb. 01, 2022. [Online]. Available: <https://samate.nist.gov/SARD/test-cases/search?language%5B%5D=c>

## BIOGRAPHIES OF AUTHORS



**Vishruti Desai**     received a B.E. degree in computer engineering from Veer Narmad South Gujarat University, Surat, Gujarat in 2002, and M.Tech. degree in 2011 from the University of Mumbai, Mumbai. Currently, she is an assistant professor at the Computer Engineering Department, C. K. Pithawala College of Engineering and Technology, Gujarat Technological University, Gujarat. She has work experience of 18 years in this domain. Her research interests include software engineering, software security, compiler design, and code analysis. She can be contacted at [vishruti.desai@ckpcet.ac.in](mailto:vishruti.desai@ckpcet.ac.in).



**Vivaksha Jariwala**     is an associate professor at Sarvajani College of Engineering and Technology. She received her M.Tech and Ph.D. from Sardar Vallabhbhai National Institute of Technology, Surat. Her research area includes information security and privacy, IoT, data science, and software engineering. She can be contacted at email: [vivaksha.jariwala@scet.ac.in](mailto:vivaksha.jariwala@scet.ac.in).