# LSAFE: a Lightweight Static Analysis Framework for binary Executables

Xiao Yue
*Department of Computer Science and Engineering*
*Oakland University*
Rochester, MI, USA
xiaoyue@oakland.edu

Guangzhi Qu
*Department of Computer Science and Engineering*
*Oakland University*
Rochester, MI, USA
gqu@oakland.edu

***Abstract*—Static analysis is a widely used technique for analyzing various aspects of programs. However, as programs become more complex, static analysis tools require larger resources, such as CPU time and memory, to perform the same tasks. Moreover, the source code of programs may not always be accessible, requiring static analysis to be performed on the binary executable code directly. To overcome these challenges, we propose a lightweight static analysis framework called LSAFE, which constructs control flow graphs (CFGs) and data dependency graphs (DDGs) of target programs with optimized performance in terms of CPU and memory usage. We evaluated the proposed framework using both Spec benchmark programs and real-world industrial applications, and found that it outperformed Angr, an existing state-of-the-art static analysis tool. Additionally, we demonstrate a case study that utilizes the CFG generated by LSAFE to detect memory leaks.**

***Index Terms*—Framework, Binary executable code, Static analysis, Control flow graph**

## I. Introduction

Static analysis [1] is a critical technique for program analysis. In contrast to dynamic analysis that requires running the programs, this method examines codes on either the source or binary level to provide information about the programs. Static program analyzers do not require the execution of the target program, making them dynamic execution-free. Consequently, they can identify all possible paths in a program, including those that will not be executed during runtime. In keeping with the principle of static analysis, most static analyzers automatically traverse the source code for analysis purposes. The literature highlights the significant role of static analyzers in enhancing software quality and security by providing a comprehensive understanding of target programs. For instance, analysis tools can be utilized to examine a C program and verify whether it adheres to API usage rules [2]. Li et al. proposed an analysis approach that generates a fine-grained fingerprint of firmware for online embedded devices [3]. As one of the primary focuses of exploiting static analysis tools, software defect detection has been explored in [4]–[6]. Static analysis techniques also assist programmers in identifying memory corruption vulnerabilities [7], detecting malware on mobile devices [8], isolating malicious behaviors [9], assisting with crash filtering implementation [10], and applying regression test selection [11]–[14]. Furthermore, with the emergence of machine learning and artificial intelligence, there is a growing trend towards integrating static analysis and machine learning to address various problems [15]–[17]. With guidance from pioneers and dedicated efforts of developers, the field of static analysis has seen rapid advancements in recent years, resulting in the development of several static analysis tools that assist developers in various aspects of their programs. Some notable examples include:

- FindBugs [18]: Proposed as analysis tool for defect detection, such as security violations and logical inconsistencies in Java programs.
- CheckStyle [19]: A static analysis tool supports multiple static analysis tasks for users, such as detection of defects.
- BAP [20]: An analysis platform with high scalability and flexibility for performing static analysis on binary code.
- PMD [21]: A static analysis tool supports many languages on finding common programming flaws.
- Firmalice [22]: It can be exploited to detect authentication bypass vulnerabilities in binary firmware.
- Angr [23], [24]: Angr supports complete dynamic symbolic execution and static binary analysis.

These tools represent significant advancements in the field of static analysis, providing developers with valuable resources for detecting defects, vulnerabilities, and inconsistencies in their code, ultimately leading to more reliable and secure software development practices. In today's increasingly complex and large-scale software landscape, analyzing programs can be resource-intensive, requiring significant CPU and memory usage. Therefore, an ideal static analyzer should offer desired features with excellent performance, without excessive resource usage. Running time is regarded as a critical metric of evaluating the performance, since a CPU time-consuming analyzer, even with additional features, may not always be preferred by users. Additionally, static analyzers are essential for supporting software development cycles, particularly for real-time embedded systems in industrial applications. However, proprietary software may not always grant access to source code, necessitating the analysis of binary executable code.

In this work, we propose a new Lightweight Static Analysis Framework for binary Executable files, noted as LSAFE,

which generates control flow graphs (CFGs) and data dependency graphs (DDGs), as well as other auxiliary data, directly from scanning disassembly files of target programs. LSAFE aims to address two challenges: 1) excessive CPU and memory usage; and 2) unavailability of source code. Specifically, we propose a novel coding mechanism to accelerate the performance of LSAFE. This mechanism optimizes operations such as determining the connection status between code blocks and handling loops in a graph. In addition to building CFGs, LSAFE also generates auxiliary data such as mapper functions and variable operations, further supporting the static analysis process.

The remaining sections of this paper are organized as follows. In Section II, we introduce the newly proposed static analysis framework, LSAFE, and details of constructed CFG. Section III presents the experimental design and performance comparison between LSAFE and Angr [24]. In Section IV, we provide a case study of utilizing the CFG built by LSAFE. Finally, we draw brief conclusions in Section V.

## II. LSAFE Architecture

This section presents a detailed description of the proposed static analysis framework, focusing on the construction of the CFGs. An overview of the LSAFE architecture is shown in Figure 1. While the disassembler plays a crucial role in translating binary representations to machine-level instructions, it is not the primary component of LSAFE. A high-quality disassembler is essential for accurate analysis, and existing works [25]–[27] have explored disassembling binary code techniques. Theoretically, LSAFE takes disassembled binary executable files generated by any disassembler from raw binary executable codes as inputs. The CFG builder constructs a CFG and other auxiliary data structures, such as variable operations and mapper functions. The DDG builder, with the assistance of mapper functions and auxiliary data created by the CFG builder, generates a DDG that represents the dependencies of variables in a program.

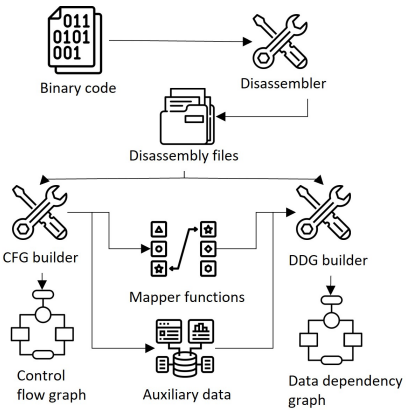


Fig. 1. LSAFE Architecture Overview

### A. Control flow graph

A CFG represents all possible execution paths of a program [28]. Constructing a CFG is a crucial step before applying further analysis to the target program, as it provides a clear view of program executions. In this work, we define a CFG as $G_{cfg} = \langle B_{cfg}, E_{cfg} \rangle$, where $B_{cfg}$ is a set of blocks, and $E_{cfg}$ is a set of edges connecting two blocks. During construction, each disassembled instruction is assigned a unique sequence number (SN). LSAFE utilizes sequence numbers to obtain the addresses of corresponding instructions, as they may not be continuous due to varying instruction sizes. Continuous sequence numbers bypass the size uncertainty of instructions and help locate the address of a specific instruction. Blocks, which are the basic elements in a CFG, contain information derived from the disassembly file of the target program. However, to minimize memory usage, LSAFE only retains the information of the first and last instructions of a block. A block has the following attributes:

- *Start Address*: This refers to the start address of a block, which is the address of the first instruction in the block.
- *Start SN*: This refers to the start SN of a block, which is the sequence number assigned to the first instruction in the block during processing.
- *End Address*: This refers to the end address of a block, which is the address of the last instruction in the block.
- *End SN*: This refers to the end SN of a block, which is the sequence number assigned to the last instruction in the block during processing.
- *Code*: This is a string code designed to check if two blocks are connected and obtain all paths between them by not traversing the graph.

Edges are the other basic elements in a CFG, connecting two blocks. Each edge has the following attributes:

- *Address of an edge*: If an edge is created for a conditional jump, an unconditional jump, or a function call, the address of the corresponding instruction is assigned to this edge. Otherwise, the address of the edge is set to *N/A*.
- *Sequence number of an edge*: If an edge is created for a conditional jump, an unconditional jump, or a function call, the sequence number of the corresponding instruction is assigned to this edge. Otherwise, the sequence number is set to *N/A*.
- *Type of the edge*: There are five different types of edges: conditional jump (CJ), unconditional jump (UJ), sequential execution (SE), function call (FC), and function return (FR). Each edge will have one of these five types.
- *Condition of the edge*: An edge has a condition if it's created due to a conditional jump. Note that two branches are created in this case. The conditional jump edge is assigned the original condition while the another branch is assigned the opposite condition denoted by *not*.
- *Source*: This represents the origin block from which the edge connects.

- *Destination*: This represents the destination block to which the edge connects.

In some existing CFG building approaches [29]–[32], all jump instructions are considered as ordinary instructions that are stored inside the blocks, and therefore the corresponding edges contain no information related to jump instructions. However, this makes it vague for users to associate the conditions and the resulting blocks in a CFG unless they check the jump instruction explicitly. In our approach, we handle jump instructions as special instructions that are stored along edges instead of in blocks. This means that edges are no longer simple pointers. They also store information about the jump instructions, which determine the execution flow from the origin block to the destination block. This allows users to have a straightforward view of how a jump instruction connects different blocks, simply by examining the information on the edges. Listing 1 and Listing 2 provide an example program called *example.c* and its disassembled machine level instructions, respectively. This example program is used to illustrate how LSAFE works.

Listing 1. An Example Program: example.c

```
1  # include <stdio.h>
2  int main(){
3    int i,j,k;
4    i=1;
5    j=2;
6    if(i>2){
7      j=3;
8      k=4;
9    } else{
10     if(j<3){
11       j=4;
12       k=5;
13     } else{
14       j=5;
15       k=6;
16     }
17   }
18   return 0;
19 }
```

Listing 2. Disassembled machine level instructions of the main function of example.c

```
00000000000005fa <main>:
5fa:    55                      push   %rbp
5fb:    48 89 e5                mov    %rsp,%rbp
5fe:    c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
605:    c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
60c:    83 7d f4 02             cmpl   $0x2,-0xc(%rbp)
610:    7e 10                   jle    622 <main+0x28>
612:    c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
619:    c7 45 fc 04 00 00 00    movl   $0x4,-0x4(%rbp)
620:    eb 24                   jmp    646 <main+0x4c>
622:    83 7d f8 02             cmpl   $0x2,-0x8(%rbp)
626:    7f 10                   jg     638 <main+0x3e>
628:    c7 45 f8 04 00 00 00    movl   $0x4,-0x8(%rbp)
62f:    c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
636:    eb 0e                   jmp    646 <main+0x4c>
638:    c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
63f:    c7 45 fc 06 00 00 00    movl   $0x6,-0x4(%rbp)
646:    b8 00 00 00 00          mov    $0x0,%eax
64b:    5d                      pop    %rbp
64c:    c3                      retq
64d:    0f 1f 00                nopl   (%rax)
```

Figure 2 depicts the CFG created by LSAFE for the example program as shown in Listings 1. Additionally, other auxiliary

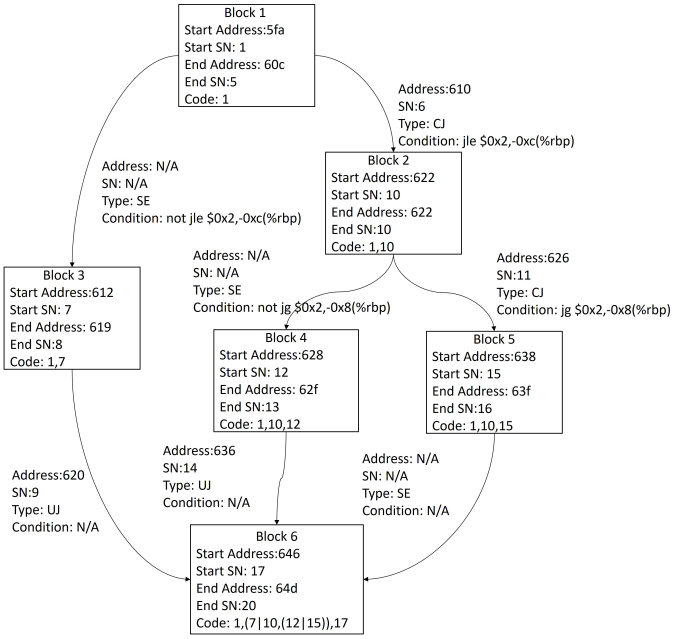


Fig. 2. CFG built from Disassembled machine level instructions file of *example.c*

data, such as variable operations, are stored in different data structures for further analysis purposes after construction of the CFG. Our framework takes a disassembled machine level instructions file as an input, which contains information about all instructions in a line-by-line format. Additionally, LSAFE also creates a mapper function to associate the sequence number of each instruction with its corresponding address, which can be used for further analysis purposes.

LSAFE employs a coding mechanism to enable efficient determination of connections between blocks and finding paths between them. To conserve memory, the graph structure is embedded into codes that are represented as strings. Each block is assigned a code during the creation of the CFG, with the *Start SN* of each block serving as a unique identifier. This method simplifies time-consuming operations, such as finding all paths between two blocks, as it eliminates the need for graph traversal. When assigning a code to a block that does not have one, the ID of this block is concatenated to the code of the previous block, separated by a comma. If a block already has a code, assigning an additional code to it indicates a path merging. In such case, the coding mechanism calculates a new code by embedding the paths using parentheses and vertical bars. Note that codes are composed of numbers separated by commas, vertical bars, or parentheses. To efficiently handle loops in the CFG, LSAFE employs a *loop table* to store paths of all loops found during the construction of the CFG, as updating codes of all blocks in a loop can be time-consuming. This coding mechanism significantly reduces time complexity of LSAFE by avoiding the need for repeated graph traversal.

### B. Data dependency graph and Auxiliary data

LSAFE builds mapper functions and other auxiliary data such as variable operations during construction of the CFG. The mapper function associates the sequence number of each instruction with its corresponding address. Variable operations track the operations performed on specific variables, recording the operation types and SN numbers of all operations. They can be exploited for constructing a DDG which represents the dependencies of variables in a program [33], as well as for conducting further analysis. LSAFE first constructs all basic blocks in a DDG using the mapper function. Subsequently, all edges in this DDG are constructed by leveraging the variable operations.

## III. Experimental Results

We conducted experiments to validate the performance of LSAFE Python implementation. We compared LSAFE with Angr, which is also primarily built with the Python programming language. Although their functions are not identical, experimental results can still provide valuable information from an academic perspective. We use a linear sweep disassembler *Objdump* as the disassembler to create disassembled files for LSAFE. The time taken to disassemble a binary code by a linear sweep disassembler is negligible compared to the running time of static analysis tools. We conducted experiments on executable programs, including Firefox, Thunderbird, and some Spec benchmark programs [34]. Firefox and Thunderbird are practical and complex open-source programs, allowing us to test the performance of LSAFE on general PC programs. Spec benchmark programs are typical computer programs with different testing purposes, which validate the scalability of LSAFE for various program types. All the experiments are conducted on the same platform with an Intel Core i7-8700 Processor, 8 GB Memory, and Ubuntu 18.04 64-bit Operating System.

### A. Statistics of resulting graphs

Table I presents statistics of CFGs built by both LSAFE and Angr. The table shows that the numbers of control blocks built by LSAFE are similar to the numbers of blocks built by Angr. However, we observed that Angr built slightly more edges due to its own algorithm. Since Angr does not provide detailed information about its algorithm for building CFGs, and visualizing large-size graphs is complicated, a direct comparison of accuracy and quality of the graphs built by the two tools is not possible at this point. This highlights the need for further research in evaluating the accuracy and quality of CFGs.

TABLE I
Number of blocks and edges in CFGs

| | Control Blocks | | Edges | |
|---|---|---|---|---|
| | LSAFE | Angr | LSAFE | Angr |
| Firefox | 54874 | 71671 | 110212 | 130080 |
| Thunderbird | 30113 | 35448 | 52094 | 59619 |
| 445.gobmk | 37518 | 36934 | 68185 | 71881 |
| 464.h264ref | 19690 | 19921 | 32699 | 36158 |
| 456.hmmer | 13965 | 13832 | 25600 | 27621 |
| 401.bzip2 | 2902 | 2959 | 4641 | 5141 |
| 429.mcf | 591 | 599 | 867 | 986 |
| 433.milc | 4994 | 4792 | 9132 | 9418 |
| 458.sjeng | 6257 | 6218 | 11044 | 12166 |
| 462.libquantum | 1804 | 1816 | 3143 | 3544 |
| 470.lbm | 372 | 402 | 554 | 627 |
| h236enc | 2357 | 2452 | 4500 | 4912 |

### B. Performance

Table II presents the performance of LSAFE and Angr in building CFGs. The running time of LSAFE is significantly less than Angr for the analysis of each testing programs. The table demonstrates that the running time of LSAFE can be approximately 2 to 5 times faster than Angr. However, it should be noted that a direct comparison of running times may not be entirely fair as the two tools may not be building identical graphs. Nevertheless, if users are specifically interested in obtaining CFGs, LSAFE proves to be a dominant choice in terms of run-time efficiency.

TABLE II
Running time (in Seconds) in building CFG

| | LSAFE | Angr | ratio (LSAFE/Angr) |
|---|---|---|---|
| Firefox | 6.95 | 19.92 | 34.88% |
| Thunderbird | 2.58 | 7.17 | 35.98% |
| 445.gobmk | 8.92 | 22.65 | 39.38% |
| 464.h264ref | 2.59 | 12.58 | 20.58% |
| 456.hmmer | 1.69 | 10.40 | 16.25% |
| 401.bzip2 | 0.38 | 1.97 | 19.28% |
| 429.mcf | 0.12 | 0.34 | 35.29% |
| 433.milc | 0.89 | 2.28 | 30.05% |
| 458.sjeng | 1.58 | 10.48 | 15.07% |
| 462.libquantum | 0.41 | 0.92 | 44.56% |
| 470.lbm | 0.12 | 0.29 | 41.37% |
| h236enc | 0.29 | 1.27 | 22.83% |

## IV. Case Study on Memory Leak Detection

Memory leaks are a common issue in software development, occurring when dynamically allocated memory is not properly released. This can result in a shortage of available memory, slow program performance, or system crashes. Detection of memory leaks can be categorized into two types: static and dynamic. In recent years, there has been significant research on static analysis methods for memory leak detection. SMOKE [35] presents a two-stage methodology for memory leak detection that balances both scalability and precision. A lightweight design called PCA [36], applies interprocedural points-to and data-flow analyses to detect memory leaks. Pinpoint [37] utilizes sparse value-flow analysis to detect memory leaks in millions of lines of code statically. CFGs are essential when analyzing memory detection problems statically as the control flow provides flow information of memory heap objects.

Based on LSAFE, memory leak detection can be effectively performed. LSAFE provides CFGs and DDGs of raw binary code, which can be utilized to identify memory allocation and deallocation points, especially in case of unavailable source code. By analyzing the flow of memory blocks in the CFG and identifying memory leak candidates, LSAFE

can aid in detecting potential memory leaks in the analyzed binary code. Additionally, LSAFE's path-sensitive solver can help filter out infeasible paths and identify path correlations based on stored conditions, further enhancing the accuracy of memory leak detection. The efficient performance of LSAFE, as demonstrated in experiments, makes it a valuable tool for memory leak detection in software development. An example program "memory_leak.c" and its disassembled machine level instructions are presented in listing 3 and listing 4 respectively.

Listing 3. An Example Memory Leak Program: memory_leak.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5    int *p = (int *)malloc(sizeof(int));
6    int a;
7    a = 1;
8    if (a>3)
9      printf("a");
10   if (a<2)
11     return 0;
12   free(p);
13   return 0;
14 }
```

Listing 4. Disassembled machine level instructions of the main function of memory_leak.c

```
0000000000001189 <main>:
1189:   f3 0f 1e fa             endbr64
118d:   55                      push   %rbp
118e:   48 89 e5                mov    %rsp,%rbp
1191:   48 83 ec 10             sub    $0x10,%rsp
1195:   bf 04 00 00 00          mov    $0x4,%edi
119a:   e8 f1 fe ff ff          callq  1090 <malloc@plt>
119f:   48 89 45 f8             mov    %rax,-0x8(%rbp)
11a3:   c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
11aa:   83 7d f4 03             cmpl   $0x3,-0xc(%rbp)
11ae:   7e 0a                   jle    11ba <main+0x31>
11b0:   bf 61 00 00 00          mov    $0x61,%edi
11b5:   e8 c6 fe ff ff          callq  1080 <putchar@plt>
11ba:   83 7d f4 01             cmpl   $0x1,-0xc(%rbp)
11be:   7f 07                   jg     11c7 <main+0x3e>
11c0:   b8 00 00 00 00          mov    $0x0,%eax
11c5:   eb 11                   jmp    11d8 <main+0x4f>
11c7:   48 8b 45 f8             mov    -0x8(%rbp),%rax
11cb:   48 89 c7                mov    %rax,%rdi
11ce:   e8 9d fe ff ff          callq  1070 <free@plt>
11d3:   b8 00 00 00 00          mov    $0x0,%eax
11d8:   c9                      leaveq
11d9:   c3                      retq
11da:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```

In this section, we define memory flow graph blocks that describe operations on memory space as follows:

- *Start of function*: These blocks indicate the start of a function and serve as starting points in a memory flow graph.
- *Memory allocation*: These blocks indicate memory allocations, typically created when the "malloc()" function is called.
- *Memory deallocation*: These blocks indicate memory deallocations, typically created when the "free()" function is called.
- *Out of scope*: These blocks indicate memory references that are out of scope, serving as end points in a memory flow graph.

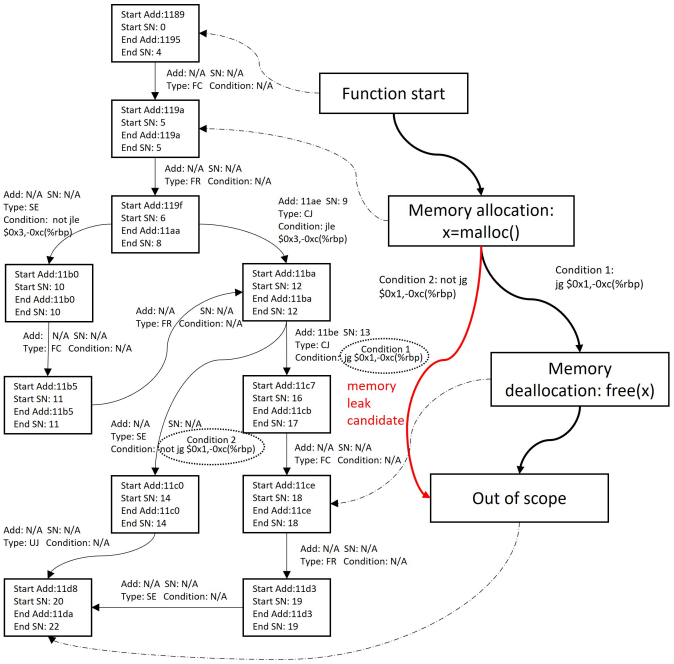


Fig. 3. CFG and memory flow graph built from example program

We constructed a memory flow graph according to the CFG of the example program using LSAFE. Figure 3 presents the CFG (left) and the memory flow graph (right) built from the example program shown in listing 3. Dashed lines indicate the locations of memory flow blocks in the CFG. Paths in the memory flow graph are created based on the CFG. In Figure 3, we have identified a potential memory leak location and labeled it as "memory leak candidate". Furthermore, since LSAFE allows for creating edges with condition information, all paths in the memory flow graph, including potential memory leak candidates, have specific conditions. By using a path-sensitive solver, we can identify all infeasible paths and path correlations based on the stored conditions in each path, which helps filter out impossible memory leak candidates. This example demonstrates that a CFG built by LSAFE provides all the necessary information for memory leak detection.

## V. Conclusions

In this paper, we have proposed a framework called LSAFE for static analysis of binary executable code. LSAFE is designed to analyze raw binary code with low resource usage, overcoming the challenge of source code absence. It leverages outputs from disassemblers and generates CFGs, DDGs, and other auxiliary data such as variable operations. We have also provided detailed of constructed CFGs such as block properties and edge properties. A novel coding mechanism that improves performance of LSAFE by avoiding unnecessary repeated operations is also proposed. Our experiments have demonstrated LSAFE's ability to construct CFGs from raw binary code with optimized performance, as validated through selected programs and compared to another state-of-the-art static analysis tool in terms of CPU and memory usage.

While differences between the two tools may affect the results, our experiments still showcase the efficiency and academic contribution of LSAFE. In addition, the case study on memory leak detection presents strong potential of LSAFE in practical applications.

## ACKNOWLEDGMENT

This material is based upon work supported by the Department of Energy Office of Cybersecurity, Energy Security, and Emergency Response (CESER) under Award Number(s) DE-CR0000023.

## REFERENCES

[1] R. E. Fairley, "Tutorial: Static analysis and dynamic testing of computer software," *Computer*, vol. 11, no. 4, pp. 14–23, April 1978.

[2] T. Ball and S. K. Rajamani, "The slam project: Debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, no. 1, pp. 1–3, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/565816.503274

[3] Q. Li, X. Feng, R. Wang, Z. Li, and L. Sun, "Towards fine-grained fingerprinting of firmware in online embedded devices," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 2537–2545.

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 234–245.

[5] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 156–165.

[6] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[7] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards locating a memory corruption vulnerability with your core dump," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 529–540.

[8] A. . Schmidt, R. Bye, H. . Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *2009 IEEE International Conference on Communications*, June 2009, pp. 1–5.

[9] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari, "Static analysis of binary code to isolate malicious behaviors," in *Proceedings. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99)*, June 1999, pp. 184–189.

[10] H. Jeon, S. Mok, and E. Cho, "Automated crash filtering using interprocedural static analysis for binary codes," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, July 2017, pp. 614–623.

[11] T. Yu, X. Qu, M. Acharya, and G. Rothermel, "Oracle-based regression test selection," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 292–301.

[12] O. Legunsen, A. Shi, and D. Marinov, "Starts: Static regression test selection," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 949–954.

[13] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 583–594.

[14] T. Yu, A. Sung, W. Srisa-an, and G. Rothermel, "Using property-based oracles when testing embedded system applications," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 100–109.

[15] M. Ijaz, M. H. Durad, and M. Ismail, "Static and dynamic malware analysis using machine learning," in *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, Jan 2019, pp. 687–691.

[16] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying android applications using machine learning," in *2010 International Conference on Computational Intelligence and Security*, Dec 2010, pp. 329–333.

[17] H. V. Nath and B. M. Mehtre, "Static malware analysis using machine learning methods," in *Recent Trends in Computer Networks and Distributed Systems Security*, G. Martínez Pérez, S. M. Thampi, R. Ko, and L. Shu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 440–450.

[18] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[19] Checkstyle. [Online]. Available: http://checkstyle.sourceforge.net

[20] A. T. S. E. Brumley D., Jager I., "Bap: A binary analysis platform," *Gopalakrishnan G., Qadeer S. (eds) Computer Aided Verification. CAV 2011.*, 2011.

[21] pmd: An extensible multilanguage static code analyzer. [Online]. Available: https://github.com/pmd/pmd

[22] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.

[23] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, Sep. 2017, pp. 8–9.

[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[25] R. N. Horspool and N. Marovac, "An Approach to the Problem of Detranslation of Computer Programs," *The Computer Journal*, vol. 23, no. 3, pp. 223–229, 08 1980. [Online]. Available: https://doi.org/10.1093/comjnl/23.3.223

[26] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.

[27] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 45–54.

[28] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[29] L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," *University of California, Davis, Tech. Rep*, 2009.

[30] K. Liu, H. B. K. Tan, and X. Chen, "Binary code analysis," *Computer*, vol. 46, no. 8, pp. 60–68, August 2013.

[31] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*, ser. AusPDC '10. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2010, pp. 61–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=1862294.1862301

[32] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy, "Interprocedural static slicing of binary executables," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, Sep. 2003, pp. 118–127.

[33] S. Horwitz, T. Reps *et al.*, "The use of program dependence graphs in software engineering," in *14th International Conference on Software Engineering: Proceedings*. Association for Computing Machinery, 1992, p. 392.

[34] "Spec benchmarks - spec.org," https://www.spec.org/contents.html.

[35] 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)*. IEEE, 2019, pp. 72–82.

[36] 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*, 2020, pp. 1621–1625.

[37] 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 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 693–706.