# Dead Code Detection Method Based on Program Slicing

Xing Wang
College of Computer
Nanjing University of Posts and Telecommunications
Nanjing, China
1015041019@njupt.edu.cn

Yingzhou Zhang*
College of Computer
Nanjing University of Posts and Telecommunications
Nanjing, China
zhangyz@njupt.edu.cn

Lian Zhao
College of Computer
Nanjing University of Posts and Telecommunications
Nanjing, China
1215043029@njupt.edu.cn

Xinghao Chen
College of Computer
Nanjing University of Posts and Telecommunications
Nanjing, China
1215043028@njupt.edu.cn

***Abstract*—Malicious code writers often insert dead code into the source code to prevent reverse engineer analysis. There are few methods for detecting dead code used in conjunction with opaque predicates. In this paper, a method based on program slicing for dead code detection is presented, which combines static slicing analysis and dynamic slicing analysis. Further, we design a dead code detection framework in the LLVM compiler infrastructure. The proposed method is used to detect the dead code inserted in some benchmark programs. The experimental results show that the proposed method has high detection rate.**

***Keywords-program slicing; dead code detection; malware analysis; LLVM infrastructure***

## I. INTRODUCTION

Dead code detection is originally a concept in compiler optimization, which has also been applied to the field of software maintenance [1], [2]. In general, dead code is code that does not affect program execution results. In order to increase the difficulty of reverse engineer analysis, malware writers often insert dead code in the program [3]. The addition of dead code makes it difficult to understand malicious code, which virtually increases the overhead of reverse analysis [4].

Srivista puts forward the static analysis method of unreachable code in object-oriented language [5], and then Jens Knoop proposes the detection method of partial dead code [6]. Yih-Farn Chen presents a data model that can be used for reachability analysis and dead code detection in C++ language [7]. Boomsma puts forward the method of eliminating dead code for web and PHP language [8]. In addition, some dead code elimination methods under integrated development tools are also presented [9], [10].

There are two deficiencies in the existing dead code detection methods. First, in malware, dead code is often used in conjunction with opaque predicates. It is difficult to detect this type of dead code because the compiler cannot infer the value of opaque predicate. Second, the dead code detection belongs to compiler optimization problem, and the existing detection methods are generally aimed at a specific programming language, which does not have general applicability.

For solving the problems above, this paper presents a method of dead code detection based on program slicing. We combine the static slicing analysis and dynamic slicing analysis [11] to detect dead code in the source code. What's more, we design a generic framework for dead code detection based on LLVM (Low Level Virtual Machine) compiler infrastructure [12].

The remaining sections of this paper are structured as follows. In section Ⅱ, we introduce the related technology. In section Ⅲ, we present the method of dead code detection based on program slicing. The framework of the dead code detection is described in section Ⅳ. In section Ⅴ, the proposed method is verified by experiments and evaluated. Section Ⅵ concludes the full paper and prospect the future work.

## II. BACKGROUND

This section briefly introduces the relevant theoretical knowledge of the proposed method, including program slicing technology and dead code theory.

### A. Program Slicing

Program slicing is a technique for simplifying and analyzing programs, which was first proposed by M. Weiser in his paper in 1979 [11]. M. Weiser found that the output of a program is related to only a portion of the program's statements and control predicates in the program. Deleting other statements and predicates does not affect program output. These statements and control predicates are called static slicing.

For decades, program slicing technology has undergone a series of developments. M. Weiser's static slicing does not consider the specific input of the program. B. Korel and J. Laski first proposed the concept of dynamic slicing. Dynamic slicing requires computing a set of statements and predicates that affect the output under a particular input. After that, more and more variants of program slicing have been proposed, such as conditional slicing, amorphous slicing, program dicing, program chopping and dynamic parallel slicing, etc. [13]. What's more, program slicing technology is also widely used in practice, which plays an important role in software analysis, software testing, software measurement, software security and so on [14].

The work was partially funded by the Open Foundation of Guangxi Key Laboratory of Trusted Software; the Qing Lan Project of Jiangsu Province.
* Corresponding author. Email addresses: zhangyz@njupt.edu.cn.

978-1-5386-2209-4/17 $31.00 © 2017 IEEE
DOI 10.1109/CyberC.2017.69

### B. Dead Code

In compiler theory, dead code contains two types of code. One can be executed, but has no effect on the result of the program, which is irrelevant code. The other may be related to the execution result of the program, but can not be executed, which is unreachable code [9].

In the source code shown in Figure 1, statement 4 and statement 6 are dead code. Statement 4 belongs to irrelevant code, for it has no effect on the execution of the program. Statement 6 is unreachable code because the value of the opaque predicate in statement 5 is always false, and statement 6 cannot be executed.

```
 1 int DeadCode (int x)
 2 {
 3    int y=0;
 4    int z=1;      // irrelevant code
 5    if (x*x < 0)
 6       y=x+1;    // unreachable code
 7    if (x*x == 0)
 8       y=y+1;
 9    if (x*x > 0)
10       y=y+2;
11    return y;
12 }
```

Figure 1. Program containing dead code

## III. Dead Code Detection Method

Program slicing plays an important role in the analysis and understanding of programs. In this section, we propose a method for detecting dead code based on program slicing. We discuss the detection of irrelevant code in section Ⅲ-A, the detection of unreachable code in section Ⅲ-B, and give an example in section Ⅲ-C.

### A. Detection of Irrelevant Code

**Algorithm 1 Detection of Irrelevant Code**

**Input:** source code P, output node set O, variable set V
**Output:** irrelevant code set C

```
S ← Ø,  C ← Ø
for each node n in O do
    S = S ∪ StaticSlice(P, n)
end for
for each variable v in V do
    if StaticSlice(P, v) not in S
        C = C ∪ StaticSlice(P, v)
    end if
end for
return C
```

The process of detecting irrelevant code is shown in algorithm 1. The input of the algorithm includes the source code P, the output node set O, and the variable set V. We first do static slicing analysis of all the nodes in the set O, and add the result to the set S. The variables in the set V are then sliced and analyzed, and the results are added to the set C if the slicing result is not in the set S. Finally, the algorithm returns the set C, which is the irrelevant code detected.

In the process of detecting irrelevant code, the set S contains all statements related to the output of the program. When slicing a variable in a program, if the slice set of this variable does not belong to a subset of S, it is shown that this variable has no effect on the results of the program. Therefore, the statements in the slice set of this variable are irrelevant code.

### B. Detection of Unreachable Code

**Algorithm 2 Detection of Unreachable Code**

**Input:** source code P, output node set O, input set I
**Output:** unreachable code set R

```
S ← Ø,  D ← Ø
for each node n in O do
    S = S ∪ StaticSlice(P, n)
end for
for each node n in O do
    for each input variable i in I do
        D = D ∪ DynamicSlice(P, n, i)
    end for
end for
R = S - D
return R
```

The process of detecting unreachable code is shown in algorithm 2. The input of the algorithm includes the source code, P, the output node set O, and the input data set I. The same as the algorithm 1, the static slicing analysis is performed on the output node set, and the results are added to the set S. Then, given different input values, the dynamic slicing analysis is performed for each output node in program P, and the results are added to the set D. Finally, we compare set S and set D, using the difference set of them as the output of the algorithm, the statements in which are unreachable code.

In algorithm 2, the static slicing set S contains all statements that may be related to the program's results, while the dynamic slicing set D contains only statements that are relevant to the results of the program execution. As a result, the statements in set S but not in set D are unreachable code.

### C. Example

We use the proposed method to detect dead code in the source code shown in figure 1. We first use algorithm 1 to detect irrelevant code. According to the slice criteria <11, y>, we perform static slicing analysis on the program, getting the statement set S = {3, 5, 6, 7, 8, 9, 10}. The variables in the variable set are then analyzed. In the program, x is the input variable, and y is the output variable. Obviously, both x and y have an impact on the results of the program execution. However, according to the slicing criterion <4, z>, we obtain the statement set {4} after static slicing analysis, which is not a subset of S. So, statement 4 is irrelevant code in the source code.

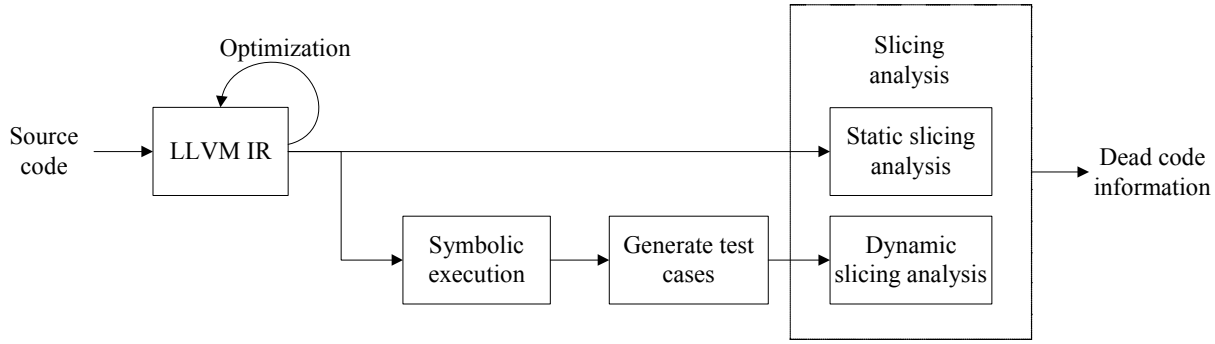


Figure 2. Framework of dead code detection

Next, we use algorithm 2 to detect unreachable code. Static analysis result has been obtained before. So we only need to carry on the dynamic slicing analysis to the source code under the different input values. When input value is 0, according to the slicing criterion <11, y, 0>, the dynamic slicing result is {3, 5, 7, 8, 9}. When input value is not 0, the result is {3, 5, 7, 9, 10}. The analysis results of the dynamic slicing statement set D = {3, 5, 7, 8, 9, 10}. So, statement set {6}, the difference set of S and D, contains unreachable code in the source code.

## IV. Dead Code Detection Framework Based on LLVM Infrastructure

In section Ⅲ, we discuss the method of dead code detection based on program slicing, which is used to detect irrelevant code and unreachable code. In this section, based on the proposed method, we give a generic framework for dead code detection.

Figure 2 shows the framework of dead code detection, which mainly consists of three parts. We first convert the source code into LLVM IR (intermediate representation), a kind of code representation in the LLVM compiler infrastructure. Then, the input test cases, which are used in dynamic analysis, are generated by symbolic execution technique. Finally, according to the detection algorithm in section Ⅲ, we analyze the LLVM IR and detect dead code in the source code.

### *A. Convert Source Code into LLVM IR*

LLVM is a compiler infrastructure used for program analysis and transformation. It provides information about compile time, link time, operation time and idle time for compiler conversion. LLVM infrastructure contains two parts: one is LLVM IR, that is, LLVM intermediate language. The other is compiler framework based on this code representation, including a series of tools related to the compiler.

In the framework of the dead code detection we design, for the sake of a unified analysis of the programs written in different languages, we first convert the source code into LLVM IR. LLVM code representation uses an abstract set of instructions in the description of program, which is similar to the RISC. It provides critical high-level information for program analysis, and makes the detection framework have general applicability.

### *B. Generate Test Cases Based on Symbolic Execution*

In the proposed detection algorithm, dynamic slicing analysis requires specific program input. Therefore, after converting source code into LLVM IR, we use symbolic execution technique to generate test cases for slicing analysis.

The main idea of symbolic execution is to use symbolic input instead of real input. When the program contains many path conditions, the simulator generates the execution tree, and each leaf node of the execution tree represents a path condition. These path conditions are sent to the constraint solver, obtaining the exact input values that can cover all execution paths.

In the next section, we use the symbolic execution tool KLEE[1]. KLEE is a symbolic execution tool under the LLVM compiler infrastructure, which can directly interpret the LLVM instruction and map it to the constraint solver. We use KLEE to analyze the LLVM IR, generating the test cases needed for slicing analysis.

### *C. Detect Dead Code by Slicing Analysis*

After converting source code into LLVM IR and generating test cases by symbolic execution, we will use the algorithm proposed in section Ⅲ to detect dead code.

Slicing analysis includes static and dynamic slicing. The static Slicing analyses the LLVM IR on the premise of not executing the program. The dynamic slicing analysis takes the generated test cases as input values, executing the program and recording the execution trace. By comparing and analyzing the results of static slicing and dynamic slicing, we can get the location of dead code in the source code.

## V. Experiment Results

Our experimental platform is a Core i5, 2.4GHz, dual core, 8GB RAM computer with 64-bit Ubuntu 14.04 operating system. We build LLVM 3.4 and KLEE on the experimental platform. We also use the self-developed slicing tool, llvm-slicing[2], and the open source dynamic slicing tool, giri[3].

[1] http://klee.github.io/
[2] https://github.com/zhangyz/llvm-slicing/
[3] https://github.com/liuml07/giri

There is no benchmark suite dedicated to dead code detection so far. We apply the dead code detection method proposed in this paper to the WCET benchmark suite [15] to ensure the correctness and generality of the algorithm. The benchmark suite is written in C language, we select some of which for experiments.

Table Ⅰ shows the insertion location of the dead code. The first column in the table is the name of programs in the WCET benchmark suite. The program that starts with D_ is the version including dead code. The second column gives the number of lines of the program. The third column gives the insertion location of the dead code. In the original program, this column is marked as not available. The fourth column gives the type of the dead code, where "Both" represents the dead code consisting of irrelevant code and unreachable code. This column is also marked as not available in the original program.

TABLE I. INSERTION LOCATION OF DEAD CODE

| Benchmark | LOC | Position of dead code | Types of dead code |
|---|---|---|---|
| Bs | 114 | N/A | N/A |
| D_Bs | 125 | 73, 88, 113 | Both |
| Duff | 170 | N/A | N/A |
| D_Duff | 178 | 109 | Unreachable code |
| Fdct | 245 | N/A | N/A |
| D_Fdct | 258 | 103, 129 | Irrelevant code |
| Prime | 47 | N/A | N/A |
| D_Prime | 52 | 41, 44 | Both |
| Recursion | 81 | N/A | N/A |
| D_Recursion | 92 | 17, 82 | Both |

We detect the dead code inserted in the benchmark suite under the framework designed in section Ⅳ. The test results are shown in table Ⅱ. The first column in the table is the name of the program that includes the dead code, which corresponds to the program name in table Ⅰ. The second column is the number of dead code rows detected. The third column is the percentage of dead code detected.

TABLE II. DETECTION RESULTS

| Benchmark | Number of dead code lines detected | Percentage detected |
|---|---|---|
| D_Bs | 11 | 100% |
| D_Duff | 7 | 87.5% |
| D_Fdct | 12 | 92.3% |
| D_Prime | 5 | 100% |
| D_Recursion | 10 | 90.9% |

The experimental results in table Ⅱ show that our method can detect the dead code inserted in the benchmark suite, with an average detection rate of over 94.1%. In addition, we can detect programs written in different languages under the proposed framework, which shows that our dead code detection method has certain versatility.

## VI. CONCLUSION AND FUTURE WORK

In this paper, a dead code detection method based on program slicing is proposed. The method combines static analysis and dynamic analysis technology to detect the dead code in the source code. In addition, we design a generic framework for dead code detection under the LLVM compiler infrastructure and check the dead code inserted into the benchmark suite.

Because of the efficiency of dynamic program slicing, we only analyze smaller programs currently. In the future, we will improve the dynamic slicing algorithm, so that our detection method can meet the needs of large-scale program analysis.

## REFERENCES


[1] Y. A. Liu and S. D. Stoller, "Eliminating dead code on recursive data," Proc. International Static Analysis Symposium, Springer Berlin Heidelberg, 1999, pp.211-231.

[2] G. Scanniello, "Source code survival with the Kaplan Meier," Proc. 27th IEEE International Conference of Software Maintenance (ICSM), 2011, pp.524-527.

[3] A.Balakrishnan and C. Schulze, "Code obfuscation literature survey," CS701 Construction of compilers, Vol. 19, 2005.

[4] S. K. Udupa, S. K. Debray and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," Proc. 12th IEEE Reverse Engineering, Working Conference, 2005, pp.45-54.

[5] A. Srivastava, "Unreachable procedures in object-oriented programming," ACM Letters on Programming Languages and Systems (LOPLAS), vol.1, no.4, pp.355-364, 1992.

[6] J. Knoop, O. Rüthing and B. Steffen, "Partial dead code elimination," Proc. ACM Sigplan 1994 Conference on Programming Language Design and Implementation, 1994, pp.147-158.

[7] Y. F. Chen, E. R. Gansner and E. A. Koutsofios, "C++ data model supporting reachability analysis and dead code detection," IEEE Transactions on Software Engineering, vol.24, no.9, pp.682-694, 1998.

[8] H. Boomsma and H. G. Gross, "Dead code elimination for web systems written in PHP: lessons learned from an industry case," Proc. 28th IEEE International Conference of Software Maintenance (ICSM), 2012, pp.511-515.

[9] H. H. Karer and P. B. Soni, "Dead code elimination technique in eclipse compiler for Java," Proc. IEEE International Conference of Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2015, pp.275-278.

[10] P. Genevès and N. Layaïda, "Eliminating dead-code from XQuery programs," Proc. 32nd ACM/IEEE International Conference on Software Engineering, 2010, pp.305-306.

[11] F. Tip, "A survey of program slicing techniques," Journal of programming languages, vol.3, no.3, pp.121-189, 1995.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," Proc. International symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Science, 2004, pp.75-86.

[13] J. Singh, P. M. Khilar and D. P. Mohapatra, "Dynamic slicing of distributed Aspect-Oriented Programs: A context-sensitive approach," Computer Standards & Interfaces, vol.52, pp.71-84, 2017.

[14] G. Negi, E. Elias, R.Kohli, et al, "Reliability analysis of test cases for program slicing," Proc. International Conference on Innovation and Challenges in Cyber Security, 2016, pp.36-40.

[15] J. Gustafsson, A. Betts, A. Ermedahl, et al, "The mälardalen WCET benchmarks: Past, present and future," Proc. International Workshop on Worst-Case Execution Time Analysis, July. 2010, pp.136-146.