Obfuscating LLVM IR with the Application of Lambda Calculus

Rei Kasuya and Noriaki Yoshiura(✉)

Department of Information and Computer Sciences, Saitama University, 255, Shimo-ookubo, Sakura-ku, Saitama, Japan
{rkasuya,yoshiura}@fmx.ics.saitama-u.ac.jp

Abstract. Software obfuscation is a method that complicates data structures and algorithms in software to prevent software from being analyzed. This paper proposes a method to obfuscate loop structures in LLVM IR (LLVM is the abbreviation of "Low Level Virtual Machine" and IR is that of "Intermediate Representation".) by applying a fixed-point combinator in the lambda calculus. LLVM IR is an intermediate representation in LLVM. A purpose of using intermediate representation in this paper is to handle multiple programming languages and architectures. This paper also evaluates the proposed obfuscation method by experiments. These experiments use loop programs, which are created artificially in this paper, and a practical program. The result of the experiments in this paper shows that the proposed method can obfuscate programs, but the execution times of the loop programs increase if these programs are obfuscated by the proposed method. However, the execution times of the practical programs do not increase under the obfuscation by the proposed method. It follows that the proposed method is available as an obfuscation method for practical programs.

Keywords: Obfuscation · LLVM IR · Lambda Calculus

1 Introduction

The advance of reverse-engineering techniques enables to analyze or modify software [1,2,9]. Reverse-engineering techniques are, however, used for plagiarism of intellectual properties from software. Software obfuscation is a method which complicates data structures and algorithms in software to prevent software from being unintentionally analyzed [8,10].

Software is classified in two types: executable files which are compiled from source codes written in programming languages and source codes which are executed by interpreters without being compiled. This paper focuses on the former kinds of software and proposes a method of obfuscation for software. Software obfuscation is classified in two types; one is to obfuscate source codes of target software before compilation and the other is to obfuscate data or machine codes in target software. Each of the two has disadvantages. The former method

N. T. Nguyen et al. (Eds.): ACIIDS 2023, LNAI 13996, pp. 27–39, 2023.
https://doi.org/10.1007/978-981-99-5837-5_3

requires a different obfuscation program for a different programming language used for the source codes of target software. For example, an obfuscation program for source codes written in C language cannot obfuscate source codes written in other programming languages. Similarly, the latter method requires a different obfuscation program for a different computer architecture because the syntax of machine codes are different according to computer architectures.

This paper proposes an obfuscation method using LLVM IR to resolve these problems. LLVM is a project to provide a base to develop compilers [5]. In particular, the LLVM Core library, a central subproject of LLVM, provides a unique intermediate representation which is called LLVM IR. A compiler using LLVM translates input source codes into LLVM IR codes and translates LLVM IR codes into machine codes of target architectures. *Intermediate representation* LLVM IR is a form of representation used in a translation process from source codes to machine codes. One purpose of using an intermediate representation is to optimize codes regardless of programming languages and machine architectures. Another purpose is that LLVM IR enables compliers to handle multiple programming languages and architectures.

The LLVM Core library also includes optimizers for LLVM IR and code generators for many architectures. Clang, a compiler developed in the LLVM project, can compile C, C++, and Objective-C. Third parties support translation tools from several programming languages into LLVM IR. For example, GHC, a DD facto standard compiler for Haskell, can translate Haskell codes into LLVM IR. Since LLVM IR is available for many programming languages and architectures, obfuscating LLVM IR codes can achieve obfuscating software written in multiple programming languages and compiled for multiple architectures.

This paper is organized as follows; Sect. 2 discusses the related works. Section 3 explains the purpose of this paper. Section 4 explains the obfuscation method of this paper. Section 5 describes the experiments and results. Section 6 discusses the result of the experiments. Section 7 concludes this paper.

2 Related Works

Pengwei et al. proposed an obfuscation method to replace integer comparison with functions applying lambda calculus [4]. Comparison operations often appear in conditional branches and therefore this method can significantly modify the control flow of the target software.

Pascal et al. proposed a method to obfuscate LLVM IR by complicating arithmetic operations, bitwise operations, the control flow, and function calls [3]. This method also implements tamper-proofing. Kyeonghwan et al. utilize the method of [3] to improve the resistance of Android applications against reverse-engineering [6]. Dongpeng et al. obfuscate LLVM IR with dynamic opaque predicates [11]. An *opaque predicate* is a complex tautology or contradiction. The most straightforward example is shown in Algorithm 1.

Although actual opaque predicates are more complex than the opaque predicate in Algorithm 1, one of the problems of opaque predicates is that static

Algorithm 1. Example opaque predicate

```
1: if 2x mod 2 = 0 then    ▷ The control will pass through this branch regardless of the value of x.
2:     ...
3: else                    ▷ The control will never pass through this branch, vice-versa.
4:     ...
5: end if
```

evaluation is possible. For example, the conditional expression $2x \bmod 2 = 0$ in Algorithm 1 is always true regardless of other program parts; thus, attackers can find the expression a tautology through static code analysis. To resolve this problem, *dynamic opaque predicates*, opaque predicates whose evaluation value might vary by each execution, have been proposed. The evaluated value of dynamic opaque predicates might vary with each execution of programs, but the programs do not change their outputs. It is difficult for attackers to detect opaque predicates.

3 Purpose

This paper proposes the method that obfuscates loop structures in LLVM IR codes by applying a fixed-point combinator in the lambda calculus. Utilizing the lambda calculus to obfuscate LLVM IR is similar to Pengwei et al [4].

Many programming languages, including C, belong to a paradigm of imperative languages. Each program has "mutable states" in imperative programming, and computers modify the states with instructions to achieve computation. LLVM IR and machine languages are also imperative languages although syntax of machine languages vary by their architecture.

Lambda calculus is a model of computation that has impacted many functional programming languages, including Haskell. The lambda calculus is calculated in a completely different way from imperative languages such as C. Thus, using the lambda calculus for obfuscation is significantly more challenging. For attackers without knowledge of the details of the proposed method, it is difficult to restore an obfuscated code to its original state.

The proposed method obfuscates loop structures in LLVM IR codes with the lambda calculus and modifies the control flow of input codes significantly. Since Pengwei et al. obfuscated integer comparison by the lambda calculus, obfuscations by the proposed method are more difficult for attackers than obfuscations by the method of Pengwei et al.

4 Method

This paper uses the lambda calculus and combinators in [7]. Because of the shortage of space, this section explains only Z-combinator. Z-combinator is widely

Algorithm 2. Calculate sum of 1 to 10 using fixed point combinator

```
1: function V(F)
2:     return
3:     function (x, i)
4:         if i < 10 then return F(x + i + 1, i + 1) else return x end if
5:     end function
6: end function
7: FIXED_POINT_COMBINATOR(V)(0, 0)
```

known as an actual instance of fixed-point combinators in the lambda calculus. The definition of the Z-combinator is as follows:

$$Z = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$$

The method proposed in this paper uses the Z-combinator implemented with C++. The proposed method first obfuscates input LLVM IR code and links it with the program implementing Z-combinator. In lambda calculus, this fixex-point combinator calculates loop program. For example, by using Z-combinator (fixed-point combinator), the function calculating the sum of 1 to 10 is given in Algorithm 2.

Next, we explains loop structures in LLVM IR. An LLVM IR code is described in the form of a control flow graph. Blocks and edges in the control flow graph are as follows;

Header block
Header block is an entrance block to a loop. Every node outside a loop has no edges with nodes other than a header block in the loop

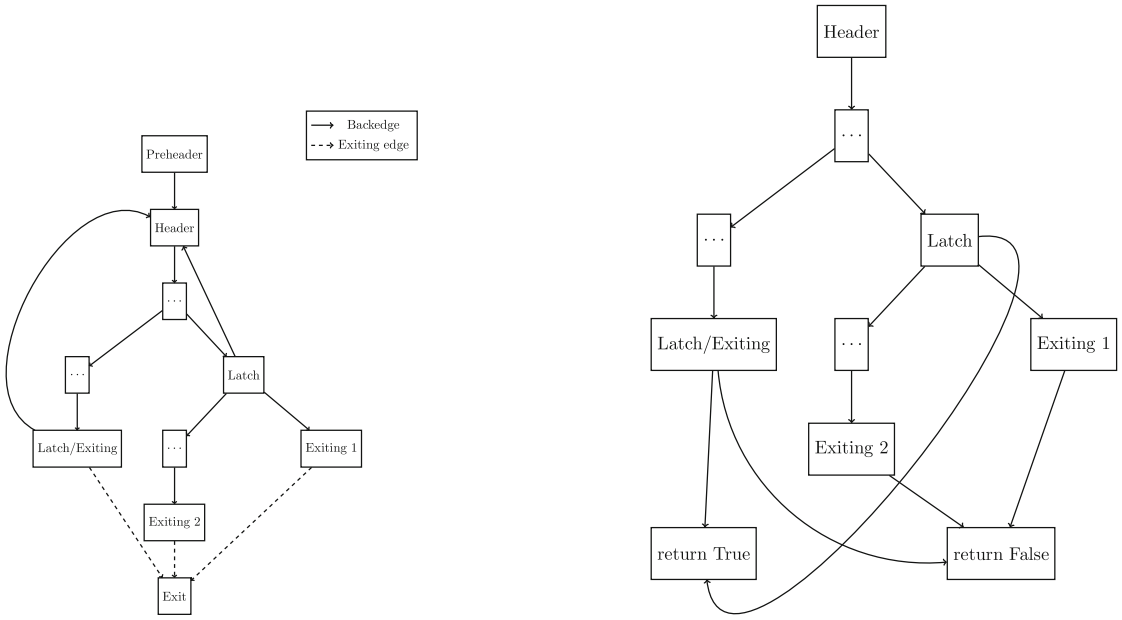
Entering blocks
Entering block of a loop is a block which is outside the loop and with edges into an header block in the loop. When a loop has only one entering block and the entering block has exactly one edge, the entering block is called preheader block.

Latch blocks
Latch block is a node in a loop and has an edge into the header block of the loop.

Backedges
Backedge is an edge from a latch block into a header block.

Exiting edges
Exiting edge is an edge from a node of a loop in a node outside the loop. The source block of an exiting edge is called exiting block and the destination block of an exiting edge is called exit block.

The proposed method obfuscates only loops that have a preheader block and exactly one exit block. However only the latter condition should be satisfied because the LLVM optimizer can modify every loop so that the loop has a preheader block.



(a) An example of a loop (b) The function extracted

Fig. 1. Obfuscating loop structure

The following explains the procedure of obfuscation by using the example loop in Fig. 1(a). Since the loop in Fig. 1 (a) has a preheader block and exactly one exit block, the loop satisfies the two conditions.

First, the procedure of obfuscation extracts instructions of the loop and defines a function for the instructions. Let this function be named "extracted." Every time the function extracted is called, it executes in-loop instructions only once and returns a truth value for representing whether the instructions should run again. More specifically, the procedure extracts blocks between the preheader block and the exit blocks, adds the blocks, "return True" and "return False" as shown in Fig. 1 (b). The created blocks consist of the function "extracted". The return value of the function "extracted" represents repeating loop or not. In the original loop structure, the control passes through a backedge when returning to the header block. Similarly, in the extracted function, the control passes through an exiting edge when exiting the loop; the return value is true when executing the loop again. The extracted function cannot refer to the variables outside the loop. If the extracted function requires to access the variables outside the loop, the variables are passed to the function extracted as arguments.

Next, the procedure of obfuscation replaces the loop by "looper function", which is described in Algorithm 3. The looper function takes, as arguments, the pointer to the extracted function and the variables required in the extracted function. The looper function, as its name suggests, executes a loop. The looper function repeatedly calls the extracted function as long as it returns "True". Algorithm 3 is the most straightforward implementation.

Algorithm 4 shows the implementation of the looper algorithm with fixed point combinator. When the loop count is enormous, a stack overflow would

Algorithm 3. Example implementation of looper function using while statement

```
1: function LOOPER(FUNCTION_TO_REPEAT, args...)
2:     whileFUNCTION_TO_REPEAT(args...) end while
3: end function
```

Algorithm 4. Example implementation of looper function using fixed point combinator

```
1: function V(F)
2:     return
3:     function (FUNCTION_TO_REPEAT, args...)
4:         if FUNCTION_TO_REPEAT(args...) then
5:             F(FUNCTION_TO_REPEAT, args...)
6:         end if
7:     end function
8: end function
9: function LOOPER(FUNCTION_TO_REPEAT, args...)
10:    Z_COMBINATOR(V)(FUNCTION_TO_REPEAT, args...)
11: end function
```

happen in Algorithm 4. Therefore, in the actual implementation of the looper function with fixed point combinator, loop execution would be switched to the while-loop algorithm like Algorithm 3 when the recursion count exceeds 8192. The pseudo-code of the actual implementation is in Algorithm 5.

5 Experiment

This section evaluates the proposed method by two experiments.

Variation with the Depth of the Loop Nesting
The looper function runs per a loop in the original code. For example, when obfuscating a double-nested loop shown in Algorithm 6, two extracted functions would be created and the looper function would run twice, as shown in Algorithm 7. The number of looper function calls increases as the nest of the loop increases; the execution time would presumably increase due to the overhead of function calls. This experiment confirms that the execution time would increase, by measuring the execution time of obfuscated programs. This experiment uses single, double, triple, quadruple and sextuple nested loop codes. Figure 2 shows only double and triple nested codes because of the shortage of space.

Each source code has a loop of different nesting depths, but the total numbers of loop iterations are the same in all of the source codes. All of the source codes execute 4096 iterations of loop. For example, the total number of loop iterations in the program of a double-nested loop is $64^2 = 4096$ and the total number of loop iterations in the program of the sextuple-nested loop is also $4^6 = 4096$.

Additionally, the numbers of loop iterations in each nest level are the same in each of the source codes. For example, in Code 1 (double-nested loop) in Fig. 2,

the number of loop iterations in each nest level is 64, and in Code 2 (triple-nested loop) inf Fig. 2, the number of loop iterations of each nest level is 16.

Algorithm 5. Implementation of looper function

```
1: function V(F)
2:     return
3:     function (recursion_count, FUNCTION_TO_REPEAT, args...)
4:         if recursion_count < 8192 then
5:             if FUNCTION_TO_REPEAT(args...) then
6:                 F(recursion_count + 1, FUNCTION_TO_REPEAT, args...)
7:             end if
8:         else while FUNCTION_TO_REPEAT(args...) do end while
9:         end if
10:    end function
11: end function
12: function LOOPER(FUNCTION_TO_REPEAT, args...)
13:    Z_COMBINATOR(V)(0, FUNCTION_TO_REPEAT, args...)
14: end function
```

Algorithm 6. Double-nested loop before obfuscation

```
1: i ← 0, j ← 0
2: while i < 10 do while j < 10 /* some calculations */ j ← j + 1 end while
3:     i ← i + 1
4: end while
```

In the experiment, the total number of loop iterations is 4096 in order to unify the total number of loop iterations in each program and the number of loop iterations in each nest level. The minimum number for the unification is 4096 because the first, second, third, fourth and sixth roots are all integers. If we use a quintuple-nested loop additionally in the experiment, the minimum number of loop iterations is 2^{60}. However, it is unfeasible to execute 2^{60} iterations. Thus, the experiment uses 4096 iterations.

The experiment consists of the three parts. The first part in the experiment measures the execution times of these loop programs which are compiled with and without the proposed obfuscation method. Every programs are run 500 times and the experimental results are the averages of 500 runs.

The second part in the experiment evaluates the performance of the proposed method quantitatively. This evaluation is based on the number of edges in the callgraph, the number of blocks and edges in the control flow graph, and the cyclomatic complexity of the control flow. The *cyclomatic complexity* of a given graph is an index of the complexity of the graph. The cyclomatic complexity is defined as $E - V + 2C$, where E is the number of edges, V is the number of vertices, and C is the number of components. Each component in a control flow graph is a graph of each individual function. The cyclomatic complexity represents the number of independent passes on the graph. The higher the cyclomatic

complexity of the control flow graph of a program is, the more complex the program structure is. Cyclomatic complexity is widely used as an indicator of software complexity and maintainability.

Algorithm 7. Double-nested loop after obfuscation

```
1: function EXTRACTED1(i, j)
2:     LOOPER(EXTRACTED2, j)
3:     i ← i + 1
4:     if i < 10 return True else return False end if
5: end function
6: function EXTRACTED2(j)
7:     /* some calculations */
8:     j ← j + 1
9:     if j < 10 return True else return False end if
10: end function
11: i ← 0, j ← 0
12: LOOPER(EXTRACTED1, i, j)
```

Code 1.1: Double-nested loop

```
int main(void)
{
  int i, j;
  int x = 0;
  for (i = 0; i < 64; ++i) {
      for (j = 0; j < 64; ++j) {
          ++x;
      }
  }
  return 0;
}
```

Code 1.2: Triple-nested loop

```
int main(void)
{
    int i, j, k;
    int x = 0;
    for (i = 0; i < 16; ++i) {
        for (j = 0; j < 16; ++j) {
            for (k = 0; k < 16; ++k) {
                ++x;
            }
        }
    }
    return 0;
}
```

Fig. 2. Source codes of the tested loop programs

The third part measures the distribution of machine instructions in the programs with and without obfuscation. If the instruction distributions significantly differ in the programs with and without obfuscation, it might be a clue for attackers to the detail of the obfuscation method. Therefore, the desired result is no significant difference in the instruction distribution in the programs with and without obfuscation.

Performance Measurement of a Practical Program

This paper also evaluates the practicality of the proposed method by implementing a program of SHA-256, which is a well-known hash algorithm, and measuring the execution times and the instruction distributions of the SHA-256 programs with and without obfuscation. The experiment results are the average time of 500 runs of the programs on random byte sequences of 100kB as an argument.

5.1 Experimental Results

Variation with the Depth of the Loop Nesting

In Table 1, *real* refers to the elapsed time from the start to the end of the program, *user* refers to the time the CPU was running in user mode during program processing, and *sys* refers to the time the CPU was running in kernel mode during program processing.

Table 1. Execution time (ms) with the depth of loop nesting

	No Obfuscation						Obfuscation					
Depth	1	2	3	4	5	6	1	2	3	4	5	6
real	5.5	5.3	5.3	5.4		5.4	13	8.2	8.3	8.5		9.4
user	4.1	4.0	3.8	4.0		4.0	6.6	5.9	6.1	6.2		6.8
sys	1.9	1.8	1.9	1.9		1.9	6.8	2.8	2.6	2.8		3.1

Table 2. Complexity indicators with the depth of the loop nesting

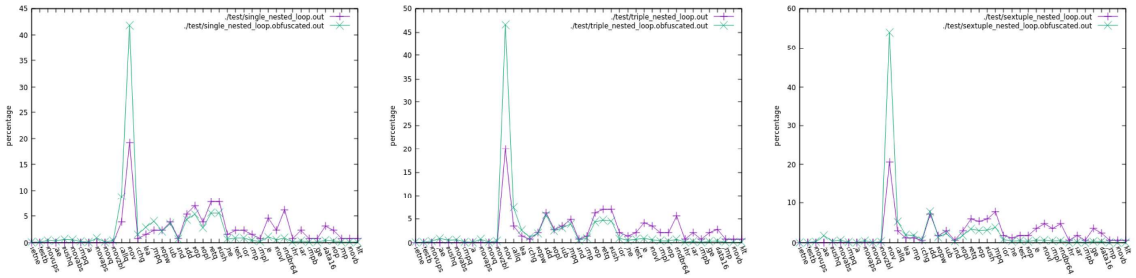
	No Obfuscation						Obfuscation					
Depth	1	2	3	4	5	6	1	2	3	4	5	6
edges in the callgraph	0	0	0	0		0	67	68	69	70		72
basic blocks	5	9	13	17		25	81	84	87	90		96
edges in the control flow graph	5	10	15	20		30	51	54	57	60		66
Cyclomatic complexity	2	3	4	5		7	72	74	76	78		82

Table 1 shows that the nesting depth does not significantly affect real, user or sys without obfuscation. On the other hand, regarding obfuscation, the single-nested loop has quite long execution times for all real, user and sys, and from the double-nested loop to sextuple-nested loop, execution times gradually increase for all real, user and sys. Table 2 shows that all four indicators significantly increase and that the proposed method increases the resistance of the programs against reverse-engineering.

Figure 3 shows the instruction distributions of the loop programs. The vertical axis of each graph is the percentage of occurrences. In the graphs, vertical crosses show the data of the programs without obfuscation and diagonal crosses show the data of the programs with obfuscation. The number of the "mov" instruction significantly increases in all nesting depths. On the other hand, the numbers of some other instructions decrease vice-versa.

Performance Measurement of a Practical Program

Table 3.(a) shows that the times of real and user increase around twofold and the time of sys remain almost unchanged. Table 3.(b) shows that all four indicators increase; this is the same as the experimental result of the loop programs. Figure 4 shows that the instruction distribution does not significantly differ with and without obfuscation.



(a) Single-nested loop (b) Triple-nested loop (c) Sextuple-nested loop

Fig. 3. Variations of the instruction distributions of the loop programs

Table 3. Experiment results of the SHA-256 program

	No obfuscation	Obfuscation
real	68 ms	140 ms
user	66 ms	130 ms
sys	23 ms	25 ms

(a) Execution time

	No obfuscation	Obfuscation
edges in the callgraph	191	258
basic blocks	283	330
edges in the control flow graph	210	220
Cyclomatic complexity	239	296

(b) Complexity indicators

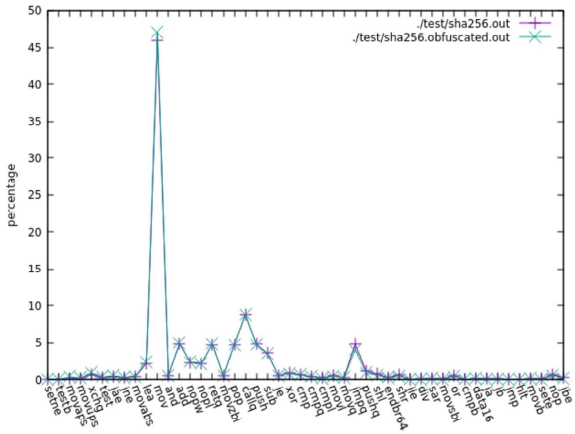


Fig. 4. Variations of the instruction distributions of the SHA-256 program

6 Discussion

6.1 Variation with the Depth of the Loop Nesting

The nesting depth did not significantly affect execution time without obfuscation in all five loop programs. However, with obfuscation, the execution time of the single-nested loop program was significantly longer than that of the other loop programs, and from the double to sextuple-nested loop program, the execution time increased gradually with the nesting depth.

We guessed that the execution time would increase as the nesting depth increased, but the measured execution time of the single-nested loop program

with obfuscation was not as expected. A candidate for the causes is a difference in the depth of function calls. For example, the function call depth of the single-nested loop reaches a maximum of 4096 since the program recurses 4096 times. On the other hand, the function call depth of the double-nested loop program reaches only a maximum of $64 + 64 = 128$. This difference in function call depth might be related to the higher than expected increase in execution time.

Next, we discuss the effects and costs of obfuscation. Obfuscation significantly increases all four complexity indicators but execution time does not increase with program complexity significantly. In particular, the execution time of the single-nested loop program has more than doubled due to obfuscation and it is necessary to reduce the increase in the execution time by improving the obfuscation method. In both methods by Pascal et al. (2015) and Pengwei et al. (2017), users can specify the percentage of how much the obfuscation is to be performed, rather than obfuscating the whole code. Since the proposed method aims to prevent reverse-engineering, it is more effective to obfuscate only the parts of the program whose processing details should not be known to third parties rather than obfuscating all loops in the program. In order to obfuscate some parts of programs, we can use source code annotations and specify the parts that should be obfuscated.

Finally, we discuss the difference in the instruction distribution with and without obfuscation. The method proposed by Pengwei et al. (2017) does not make a significant difference in the instruction distribution with and without obfuscation. Making no difference is the reason why an attacker presumably cannot know the details of obfuscations. In contrast, the proposed method makes non-negligible differences in the instruction distribution; for example, the proposed method makes significant differences in the occurrence rates of "mov" instruction. These differences may provide clues for an attacker who would like to analyze programs by reverse-engineering.

6.2 Performance Measurement of a Practical Program

We discuss the result of experiment of a practical program, SHA-256 program. The four complexity indicators increased in the obfuscated SHA-256 program, but the execution time also nearly doubled. As mentioned in Sect. 6.1, specifying the parts which should be obfuscated in the programs may prevent an increase in the execution time of an obfuscated program.

Compared with the experiment of the loop programs, instruction distributions between the SHA programs with and without obfuscation are not significantly different. This result is an advantage of the proposed method.

This discuss indicates that for a small program consisting such loop programs, a difference in instruction distribution is different before and after obfuscation. However it is expected that the difference is unnoticeable when the sizes of programs exceed a certain level.

7 Conclusion

This paper proposed the method which obfuscates loop structures in LLVM IR by applying a fixed-point combinator in the lambda calculus. As a result, the complexities of the programs significantly increase, but the increase in the execution time of the program is not negligible. However specifying parts which should be obfuscated may resolve this problem.

An instruction distribution is significantly different between programs with and without obfuscation if the programs are relatively small. For programs whose sizes are relatively large, a significant difference before and after obfuscation does not appear. Since practical programs are relatively large, the proposed method does not have a disadvantage for practical programs.

Future work includes the development of more effective obfuscation methods, as described in Sect. 6, in order to reduce the increase in execution time. Another future work is to propose another obfuscation methods for other control structures such as conditional branches.

References

1. Bhardwaj, V., Kukreja, V., Sharma, C., Kansal, I., Popali, R.: Reverse engineering-a method for analyzing malicious code behavior. In: 2021 International Conference on Advances in Computing, Communication and Control (ICAC3), pp. 1–5 (2021)
2. Jain, A., Gonzalez, H., Stakhanova, N.: Enriching reverse engineering through visual exploration of Android binaries. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5), pp. 1–9 (2015)
3. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM - software protection for the masses. In: Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection, SPRO 2015, pp. 3–9 (2015)
4. Lan, P., Wang, P., Wang, S., Wu, D.: Lambda obfuscation, security and privacy in communication networks. In: 13th International Conference, SecureComm 2017, Proceedings, pp. 206–224 (2017)
5. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of International Symposium on Code Generation and Optimization, pp. 75–86 (2004)
6. Lim, K., et al.: An anti-reverse engineering technique using native code and obfuscator-LLVM for Android applications. In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS 2017, pp. 217–221 (2017)
7. Roger Hindley J., Seldin, J.P.: Lambda Calculus and Combinators: An Introduction (2nd. ed.). Cambridge University Press, Cambridge (2008)
8. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Comput. Surv. **49**(1), 1–37 (2016)
9. Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L.: An observational investigation of reverse engineers' processes. In: The Proceedings of 29th USENIX (2020)

10. Kang, S., Lee, S., Kim, Y., Mok, S.K., Cho, E.S.: OBFUS: an obfuscation tool for software copyright and vulnerability protection. In: Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY 2021), pp. 309–311 (2021)
11. Xu, D., Ming, J., Wu, D.: Generalized dynamic opaque predicates: a new control flow obfuscation method. In: 19th Internation Conference on Information Security, pp. 323–342 (2016)