Digital Watermarks for LLVM Intermediate Representation

Ryoga Nagayama
Kyoto Institute of Technology
Kyoto, Japan
nagayama15@sec.is.kit.ac.jp

Lu Chen
Kyoto Institute of Technology
Kyoto, Japan
l-chen@kit.ac.jp

Hiroyuki Inaba
Kyoto Institute of Technology
Kyoto, Japan
inaba@kit.ac.jp

***Abstract*—We propose several methods to embed digital watermarks in LLVM intermediate representation. Digital watermarks are embedded by modification of LLVM IRs. These methods realize digital watermarks that are independent of project's development language and its target platforms.**

***Index Terms*—Digital Watermark, Software Watermark, LLVM, Intermediate Representation**

I. Introduction

In recent years, with the development of the Internet, it is easy to transmit digital contents including software. On the other hand, uploading and downloading of contents, ignoring copyright is a problem. Digital watermarking that embeds information in digital contents attracts attention as a solution to this problem.

Digital watermarking is a branch of steganography that encodes information inside the data. Although digital watermarking can not restrict unauthorized copying of content itself, it is possible to identify the user who has illegally uploaded the content by reading user identification information embedded in the content.

Unlike the medium of such as video, audio, and image, data hiding in executable code has not been the subject of much study. In digital watermarking for such multimedia content, information is usually embedded using content redundancy.

In the proposed method, by embedding digital watermarks in LLVM intermediate representation (LLVM IR), it becomes possible to embed a digital watermark independently of a specific programming language or machine code.

II. Related Works

There are two basic approaches of embedding a digital watermark in software.

One is to embed the watermark by modifying the program binary directly. El-Khalil et al. [1] proposed a method for embedding information using the equivalent instructions of x86 machine code. There are also studies to embed watermarks in Java bytecode [2] [3].

The other approach is to embed the watermark by modifying the source code. Fionov proposed a method to embed watermarks by changing the order of independent statements and variable declarations present in the source code of the C/C++ project [4]. However, it does not consider resistance to optimization, and furthermore, due to the language features, only very small embedding capacity can be obtained for actual C++ projects.

Also, Monden et al. [5] proposed a method combining the change of Java program source code and the change of Java bytecode. It adds a dummy method that is not actually executed as an area for embedding watermarks in a Java class, and embeds information in the instructions or operands inside the method. This method has the advantage of being able to embed watermarks of arbitrary length by making a larger size of dummy method, and to be resistant to simple obfuscation in which symbol names are changed. However, as described in [6], the embedding location can be easily identified from the unnaturalness of the decompilation result. Furthermore, since the instructions in the dummy method are irrelevant to the behavior of the actual program, the content can be easily rewritten if the embedding location is known [7].

Also, these existing methods depend on the characteristics of the programming language and/or the format of machine code.

III. Structure of LLVM Intermediate Representation

LLVM [8] is a compiler infrastructure that supports native code output to various target platforms. In addition, it provides optimization for intermediate representations and machine code.

Intermediate expressions are data structures that the compiler uses internally to analyze and optimize the program structure. Intermediate representations are intended to share code generation between different programming languages. Besides, it is also used for program analysis and optimization. LLVM IR, an intermediate representation provided by LLVM, consists of multiple abstracted units such as modules, functions, basic blocks and instructions.

A module is the top-level object in the LLVM IR and it corresponds to a translation unit, i.e. a single source file. Modules have definitions and declarations of constants, global variables, functions, and meta information about the program.

A function is an element that corresponds to a procedure in the program. It can have multiple inputs (arguments) and one output (a return value). The processing inside the function is expressed as a graph of basic blocks.

978-1-7281-3575-5/19/$31.00 ©2019 IEEE
Authorized licensed use limited to: Universitatsbibliothek Erlangen Nurnberg. Downloaded on January 05,2021 at 15:26:00 UTC from IEEE Xplore. Restrictions apply.

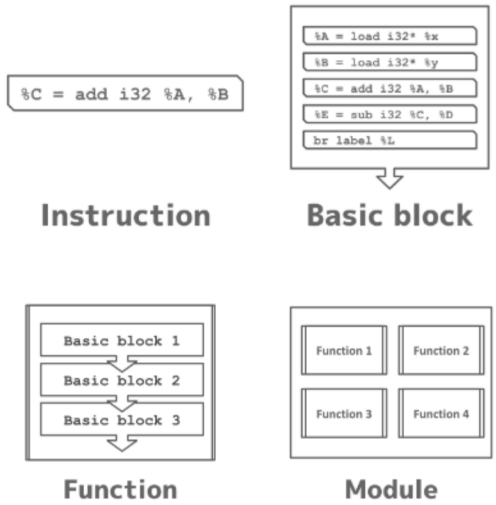

Fig. 1. The elements of LLVM IR.

Basic blocks that make up a function are chunks of instructions. Every basic block in a program ends with a terminator instruction. It indicates which basic block should be executed after the current basic block is finished. Terminator instructions include `ret` for returning from the current function and `br` for conditional and unconditional jumps.

Most instructions are represented in a form similar to machine language, and indeed their behavior is also similar, but not dependent on a particular architecture.

LLVM also provides powerful optimizations for programs. They are performed at three levels.

- Intermediate representation level optimization
- Machine code optimization
- Link time optimization

Intermediate representation level optimization produces more efficient LLVM IR by modifying its structure. Each optimizer, such as the reduction of the number of instructions and the inline expansion of functions, is provided by a small object called "pass", and combining them realizes higher optimization.

Machine code optimization performs optimization by changing the native machine code. Machine code is represented by the LLVM machine intermediate representation (LLVM MIR), which is some sort of intermediate representation similar to the LLVM IR. The difference between the two is whether it depends on the target architecture. For example, an IR can have an infinite number of virtual registers, but an MIR can have registers as few as the actual instruction set.

The two optimizers above work only in one module. However, link time optimization performs beyond translation units at the time of object file linking. This makes it possible to perform control flow analysis, etc. that takes global variables into consideration, which is usually difficult. However, even though this does not contribute so much to the execution speed, it causes an increase in compile time and an increase in the amount of data of the object file, so it is invalidated unless an option is explicitly specified.

IV. Implementing Embedding Methods for LLVM Intermediate Representation

Embedding is realized in the following three ways.

- Method-1: Changing the order of basic blocks.
- Method-2: Swapping instruction operands.
- Method-3: Changing the order of functions.

Method-1: Changing the order of basic blocks.

The first method embeds digital watermarks by changing the order of basic blocks contained in the same function. Basic blocks are always terminated by a jump instruction such as `br` and `ret`. Even if the order of basic blocks other than entry blocks (the first basic block in a function) is changed, the behavior of the function does not change. In this method, digital watermarks are embedded as a change in the instruction sequence of the machine code corresponding to the basic block.

For example, there is a simple source code snippet shown in Fig. 2 written in C. It is compiled into the LLVM IR like Fig. 3. This IR consists of 4 basic blocks. Since the first one is an entry block of a function, changing the order of the other 3 basic blocks does not compromise the semantics of the program. Embedding a digital watermark in the LLVM IR uses this method will generate an IR like Fig. 4. Assembling Fig. 3 and Fig. 4 into x86_64 machine code, Fig. 5 and Fig. 6 are obtained.

```
return a % 2 ? x : y;
```

Fig. 2. A simple source code written in C.

```
  %5 = srem i32 %4, 2
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %9

; <label>:7
  %8 = load i32, i32* %2, align 4
  br label %11

; <label>:9
  %10 = load i32, i32* %3, align 4
  br label %11

; <label>:11
  %12 = phi i32 [ %8, %7 ], [ %10, %9 ]
  ret i32 %12
```

Fig. 3. The compilation result LLVM IR.

The capacity of information that can be embedded in this method is determined by the number of basic blocks included in the function. Let $N_{f,BB}$ be the number of basic blocks included in the function f, the capacity of watermarks that can be embedded in the function f is $\lfloor \log_2 (N_{f,BB} - 1)! \rfloor$ bit.

Method-2: Swapping instruction operands.

Some of LLVM IR instructions are represented in a form similar to the 3-address code. That is, it has 2 input operands and one output. Some of these binary instructions, such as

Authorized licensed use limited to: Universitatsbibliothek Erlangen Nurnberg. Downloaded on January 09,2021 at 19:28:00 UTC from IEEE Xplore. Restrictions apply.

```
  %5 = srem i32 %4, 2
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %11, label %7

; <label>:7:
  %8 = load i32, i32* %3, align 4
  br label %9

; <label>:9:
  %10 = phi i32 [ %12, %11 ], [ %8, %7 ]
  ret i32 %10

; <label>:11:
  %12 = load i32, i32* %2, align 4
  br label %9
```

Fig. 4. The LLVM IR with a digital watermark embedded in method-1.

```
    idiv esi
    cmp  edx, 0
    je   .LBB0_2
#.LBB0_1:
    mov  eax, [rbp-8]
    mov  [rbp-20], eax
    jmp  .LBB0_3
.LBB0_2:
    mov  eax, [rbp-12]
    mov  [rbp-20], eax
.LBB0_3:
    mov  eax, [rbp-20]
    pop  rbp
    retq
```

Fig. 5. The x86_64 machine code.

add, imul and xor, are "commutative". That means the output will not change even if the order of the operands changes. The second method embeds digital watermarks by swapping two operands of a commutative binary instruction.

Also, the comparison instruction icmp and fcmp can swap the two operands by changing the direction of the inequality sign as shown in Fig. 7.

The capacity of information that can be embedded in method-2 depends on the number of commutative binary instructions and comparison instructions. Let $N_{f,Op}$ be the number of such instructions in the function f, the capacity of watermarks that can be embedded in the function f is $N_{f,Op}$ bit.

```
    idiv esi
    cmp  edx, 0
    jne  .LBB0_1
#.LBB0_2:
    mov  eax, [rbp-12]
    mov  [rbp-20], eax
.LBB0_3:
    mov  eax, [rbp-20]
    pop  rbp
    retq
.LBB0_1:
    mov  eax, [rbp-8]
    mov  [rbp-20], eax
    jmp  .LBB0_3
```

Fig. 6. The x86_64 machine code with a digital watermark embedded in method-1.

```
; c = a + b
%c = add i32 %a, %b
; c = b + a (Same as above)
%c = add i32 %b, %a

; z = x < y
%z = icmp slt i32 %x, %y
; z = y > x (Same as above)
%z = icmp sgt i32 %y, %x
```

Fig. 7. Example of swapping the operands of LLVM binary instructions.

Method-3: Changing the order of functions.

Method-3 embeds by changing the insertion order of functions in the module. When the insertion order is changed, corresponding functions in the machine code are also output in the same order. Watermarks are represented by permutations of functions.

By manipulating the LLVM IR, we can freely change the order of the functions defined in the module, but exchanging functions at a distant location in the module may cause cache misses. Therefore, changing the order of functions should be limited within adjacent groups of functions.

Let G_i $(i = 1, \ldots, n)$ be a group of functions in the module, and N_{G_i} be the number of functions exists in the group G_i. The capacity of watermark that can be embedded in the module is $\sum_{i=1}^{n} \lfloor \log_2 N_{G_i}! \rfloor$ bit.

Since each of these embedding methods is independent of the others, we can apply multiple methods simultaneously.

V. Resistance to Optimization

All methods embed digital watermarks into IRs. Therefore, the effect of intermediate representation level optimization can be ignored by embedding in the optimized IR. However, machine code optimization and link time optimization may destroy the watermark information embedded by the methods described above. In this chapter we will consider watermark tolerance for these optimizations.

Machine code optimization attempts to replace instructions or relocate basic blocks to improve execution efficiency. Therefore, some watermarks embedded by method-1 and method-2 are lost by this optimization. In contrast, method-3 is resistant to machine code optimization because no function relocation is performed.

Link time optimization performs inline expansion of functions and analysis of global variables beyond translation units. Since it sometimes involves decreasing functions or changing control flow, watermarks embedded in any method can be lost with this kind of optimization.

VI. Performance Evaluation

We evaluate the capacity of watermark information that can be embedded using the proposed methods and the effect on execution speed.

Data Rate

We measure the capacity of watermark information that can be embedded in several open source C/C++ projects published on the Internet. Watermarks are embedded according to the following procedure.

1) Output LLVM IR from source code using Clang [9], a LLVM based C/C++ compiler.
2) Embed watermarks in the output LLVM IR.

Authorized licensed use limited to: Universitatsbibliothek Erlangen Nurnberg. Downloaded on January 09,2021 at 19:28:00 UTC from IEEE Xplore. Restrictions apply.

3) Generate an object file from the IR.
4) Link them to create an executable.

For each method, the measurement results of the capacity of information embedded are shown in Table I, Table II and Table III.

TABLE I
MEASUREMENT RESULT OF THE CAPACITY OF WATERMARK INFORMATION EMBEDDED BY METHOD-1

Project	Code size [KB]	The capacity of watermark [bit] (without optimization)	The capacity of watermark [bit] (with IR level optimization)
tree	98.6	2808	2208
cJSON	110.8	1908	1596
8cc	200.6	3888	4452
zlib	307.9	4776	3840
jemalloc	625.2	57096	19524
lua	665.8	8016	11556

TABLE II
MEASUREMENT RESULT OF THE CAPACITY OF WATERMARK INFORMATION EMBEDDED BY METHOD-2

Project	Code size [KB]	The capacity of watermark [bit] (without optimization)	The capacity of watermark [bit] (with IR level optimization)
tree	98.6	1197	1285
cJSON	110.8	807	1212
8cc	200.6	1154	1819
zlib	307.9	2537	3511
jemalloc	625.2	16135	14600
lua	665.8	4831	7926

TABLE III
MEASUREMENT RESULT OF THE CAPACITY OF WATERMARK INFORMATION EMBEDDED BY METHOD-3

Project	Code size [KB]	The capacity of watermark [bit] (without optimization)	The capacity of watermark [bit] (with IR level optimization)
tree	98.6	80	79
cJSON	110.8	289	213
8cc	200.6	1022	368
zlib	307.9	265	200
jemalloc	625.2	3153	1398
lua	665.8	2037	1167

For method-1 and method-2, in some projects, IR level optimization reduced the capacity of watermark information that can be embedded. This is considered to be due to redundant branches and some instructions removed by optimization. On the other hand, the capacity has increased for other projects. This is because functions are inlined, and the number of basic blocks and instructions contained in the program has increased.

Regarding method-3, the capacity has been reduced by IR level optimization for all projects. This is also an effect of inline expansion. This is because the number of functions contained in the program is reduced, and the number of permutations that can be expressed in the program is reduced.

Execution Speed

We experimented about the influence on execution speed by the embedding of digital watermarks using the method-3.

The execution time was measured by the following procedure.

1) Apply IR level optimization to the LLVM IR of the project zlib.
2) Embed digital watermarks on them using the method-3.
3) Generate an executable with machine code optimization.
4) Compress a 200MB fixed random number sequence using `deflate` function in zlib and measure its execution time.

We measured 100 times for each executable with different digital watermarks embedded. The experimental environment is as follows:

- OS : Ubuntu 18.04
- CPU : Intel Core i5-6600K
- Compiler : Clang 7.0.0

TABLE IV
MEASUREMENT RESULT OF EXECUTION TIME

Embedded watermark	Average execution time $\bar{t}$ [ms]	σ [ms]
No watermark	4425	49
92bb9dcfd3b9f053	4434	34
48d92373664b84ba	4412	35
28e2fb60e8dc7c19	4406	31

As shown in Table IV, embedding of digital watermarks had little impact at execution time.

We also measured the difference in execution speed due to link time optimization for the executables without watermarks using the same procedure.

TABLE V
THE IMPACT OF LINK TIME OPTIMIZATION ON EXECUTION TIME.

	Average execution time $\bar{t}$ [ms]	σ [ms]
No link time optimization	4425	49
Link time optimized	4412	40

As shown in Table V, with this type of program, the impact of link time optimization was negligible.

VII. APPLICATION TO DIFFERENT LANGUAGES AND PLATFORMS

LLVM is used as a compiler backend with many compilers other than Clang. Some of them can output LLVM IR from source code as well as Clang. Thus, even software written in a programming language other than C/C++, watermarks can be embedded using the proposed method, if the compiler can output LLVM IRs.

For example, there is a small program shown in Fig. 8 written in Rust programming language [10].

Compiling it with rustc, the most famous Rust compiler implementation, outputs LLVM IR as shown in Fig. 9.

Authorized licensed use limited to: Universitatsbibliothek Erlangen Nurnberg. Downloaded on January 09,2021 at 19:28:00 UTC from IEEE Xplore. Restrictions apply.

```
fn main() {
    (1 ..= 50).for_each(|i| match (i % 3, i % 5) {
        (0, 0) => println!("Fizz Buzz"),
        (0, _) => println!("Fizz"),
        (_, 0) => println!("Buzz"),
        (_, _) => println!("{}", i),
    });
}
```

Fig. 8. A small program written in Rust programming language.

```
; fizzbuzz::main
; Function Attrs: nonlazybind uwtable
define internal void @_ZN8fizzbuzz4main17hbacf013637042430E()
unnamed_addr #0 personality i32 (i32, i32, i64,
%"unwind::libunwind::_Unwind_Exception"*,
%"unwind::libunwind::_Unwind_Context"*)* @rust_eh_personality {
start:
    %_48.i.i.i.i.i.i = alloca [1 x { i8*, i8* }], align 8
    %_41.i.i.i.i.i.i = alloca %"core::fmt::Arguments", align 8
    %_33.i.i.i.i.i.i = alloca %"core::fmt::Arguments", align 8
    %_25.i.i.i.i.i.i = alloca %"core::fmt::Arguments", align 8
    %_17.i.i.i.i.i.i = alloca %"core::fmt::Arguments", align 8
    %i.i.i.i.i.i.i = alloca i32, align 4
    %0 = bitcast i32* %i.i.i.i.i.i.i to i8*
    %1 = bitcast %"core::fmt::Arguments"* %_17.i.i.i.i.i.i to i8*
    ; (omit)
}
```

Fig. 9. Output LLVM IR of the Rust program.

Watermarks can be embedded in this LLVM IR and we can obtain a watermarked executable file by compiling it with Clang.

Because LLVM IRs do not rely on specific output targets, it is also possible to generate a watermarked executable for different targets other. Assembling Fig. 3 and Fig. 4 into ARM machine code, Fig. 10 and Fig. 11 are obtained.

```
    cbnz w8, .L0
    b .L1
.L0:
    ldr x0, [sp, #16]
    ldr x8, [sp, #8]
    blr x8
    b .L2
.L1:
    ldr x0, [sp]
    ldr x8, [sp, #8]
    blr x8
.L2:
    mov w0, #0
    ldp x29, x30, [sp, #32]
    add sp, sp, #48
    ret
```

Fig. 10. The ARM machine code.

```
    cbnz w8, .L0
#.L1:
    ldr x0, [sp]
    ldr x8, [sp, #8]
    blr x8
.L2:
    mov w0, #0
    ldp x29, x30, [sp, #32]
    add sp, sp, #48
    ret
.L0:
    ldr x0, [sp, #16]
    ldr x8, [sp, #8]
    blr x8
    b .L2
```

Fig. 11. The ARM machine code with a digital watermark embedded in method-1.

VIII. Conclusion

The proposed methods embed digital watermarks by modification of LLVM IR. Watermarks are embedded by operations such as change order of basic blocks, swapping the operands of commutative binary instructions and comparison instructions, and change insertion order of functions.

LLVM IR does not depend on a specific programming language or machine code. Therefore, this embedding method is also independent of them. In particular, method-3 is resistant to machine code optimization, so execution speed does not become slow. Efficient watermark decoding method is a future work.

Acknowledgment

I would like to express my gratitude to my family for their moral support and warm encouragements.

References

[1] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding Information in Program Binaries," International Conference on Information and Communications Security, October 2004

[2] A. Monden, H. Iida, "A watermarking method for computer programs" (in japanese), In Proceedings of the 1998 Symposium on Cryptography and Information Security, SCIS'98. Institute of Electronics, Information and Communication Engineers, January. 1998.

[3] A. Monden, H. Iida, K. ichi Matsumoto, K. Inoue, and K. Torii, "Watermarking java programs," In International Symposium on Future Software Technology ' 99, pages 119–124, October 1999.

[4] A. Fionov, "Digital Watermarks for C/C++ Programs," 7TH FRUCT, 2009

[5] A. Monden, H. Iida, K. Matsumoto, K. Torii and K. Inoue, "A Practical Method for Watermarking Java Programs," 24th International Computer Software and Applications Conference, January 2000

[6] K. Fukushima, T. Tabata, T. Tanaka, K. Sakurai, "Embedding user identification information in Java program using class structure conversion" (in japanese), Information and Media Technologies, Vol. 46, No. 8, pp. 2042–2052, August 2005.

[7] K. Fukushima, K. Sakurai, "Obfuscating of Embedding Position of the Software Fingerprinting," The 2003 Symposium on Cryptography and Information Security, The Institute of Electronics, Information and Communication Engineers. January 2003

[8] The LLVM Compiler Infrastructure Project, https://llvm.org/

[9] Clang C Language Family Frontend for LLVM, https://clang.llvm.org/

[10] Rust Programming Language, https://www.rust-lang.org/

Authorized licensed use limited to: Universitatsbibliothek Erlangen Nurnberg. Downloaded on January 09,2021 at 19:28:00 UTC from IEEE Xplore. Restrictions apply.