# *Three Methods of Control Flow Obfuscation on Java Software*

Ming-Hsiu Tsai, Hsiang-Yang Chen, Ting-Wei Hou
*Department of Engineering Science,*
*National Cheng Kung University, Republic of China*
mingshu@nc.es.ncku.edu.tw i14248@mail.hku.edu.tw hou@nc.es.ncku.edu.tw

***Abstract***-*A defense against reverse engineering is obfuscation, a process that renders software unintelligible but still functional. Our goal is to let all known decompilation techniques fail to decompile Java programs and lower the re-engineering level to assembly language (bytecode). We design three new obfuscation methods for protecting Java code. Our new designed techniques are named as "destroying basic block obfuscation", "replacing goto obfuscation" and "intersecting loop obfuscation". We use 16 different available decompliers to examine and compare our obfuscations. As the result, both the "replacing goto obfuscation" and the "intersecting loop obfuscation" could succeed to defeat all the decompilers.*

***KEYWORD*****:** reverse engineering attacks, obfuscation, Java, decompiler

## 1. INTRODUCTION

Since the Java decompiler appeared [1], the threat of reverse engineering becomes worth-noticing. The Java language was designed to compile into a platform being independent bytecode format. Much of the information contained in the source code remains in the bytecodes, which means that the decompilation is easier than the traditional native code. Today, it is not a secret that Java programs can be easily decompiled and reverse-engineered from Java bytecode to Java source code [2].

A defense against reverse engineering is obfuscations. Obfuscation is a process that it keeps the program's semantics but makes the program difficult to decompile. The design of obfuscations is to prevent from the theft of intellectual property by making it unable to derive usable source code from bytecode. Obfuscating transformations can be applied automatically to a program by a tool called an obfuscator. Using the obfuscator to protect intellectual property for Java commercial software is very important. Obfuscations have become a critical to commercial software licensing.

In figure 1-1,the types of obfuscation techniques are as follows [3] [4]:

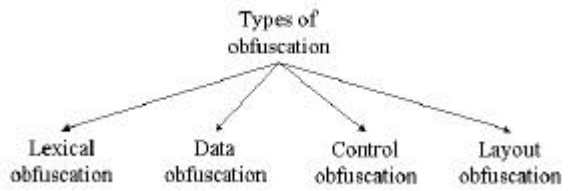


***Figure 1-1. Types of obfuscations.***

*Lexical obfuscations* modify the lexical structure of the program. Typically, they do nothing more then scramble identifiers. All meaningful symbolic information of a Java program, such as classes, fields, and method names are replaced with meaningless information, such as Crema [5] Java obfuscator.

*Data obfuscations* modify the program data fields. For example, it is possible to replace an integer variable in a program with two integers.

*Control obfuscations* make thieves difficult to understand the control flow in individual program functions [6][7]. One example, the opaque predication, uses conditional instructions whose predication always evaluates true or false. The branch of such a condition that is always taken will contain a meaning code, while the other branch will contain an arbitrary code.

*Layout obfuscations* involve obscuring the logic inherent in splitting a program into procedures. One approach is to perform in-line expansion of a procedure in all places where the procedure is called.

### 1.1 Control Flow Obfuscation categories

Our study was about control obfuscations. Figure 1-2 introduces their categories [7].

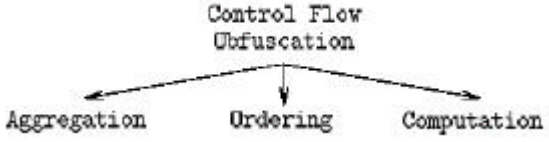


***Figure 1-2. The categories of Control flow obfuscation.***

*Control Aggregation obfuscations* change the way in

*1*

which program instructions are grouped together. Inlining and outlining are one of the most effective ways in which methods and method invocations can be obscured.

*Control ordering obfuscations* change the order in which instructions are executed. For example, loops can sometimes be made to iterate backwards instead of forwards.

*Control computation obfuscations* hide the real control flow in a program. For example, instructions that have no effect can be inserted into a program.

### 1.2 Control Computation Flow Obfuscation

Control computation obfuscations fall into the three categories in Figure 1-3 [7].

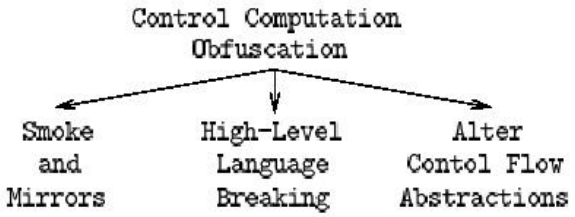


***Figure 1-3. Control computation obfuscation categories.***

*Smoke and Mirrors obfuscations* are to hide the real control flow behind instructions that are irrelevant. Inserting dead code into a program is an example.

*High-Level language breaking obfuscations* introduce features at the object code level that has no direct source code equivalent. For example, Java does not have the goto statement. Inserting goto instructions at bytecode level can make decompilers unable to find suitable flow graphs.

*Alter control flow obfuscation* is the process of taking a sequence of low-level instructions and forming an equivalent description at a higher level. It can remove abstraction from the program. For example, a for-loop in the C language source code can be changed into an equivalent loop that uses "if" and "goto" statements.

## 2. Design methods

Now, most obfuscators on the market adopt lexical obfuscation. The lexical obfuscated programs still can be decompiled to high-level language easily. Reverse engineering attacks for dealing with low-level language is harder, so software will be protected better. Therefore, the goal of our approach is to have some new control flow obfuscation techniques that the decompilers cannot decompile obfuscated programs. The attacker will not get the correct Java source code.

We first discuss the notion of the opaque predicate, which is an important element of many control flow obfuscations. As figure 2-1 shows, if its outcome is known false at execution time, an opaque predication is F. If its outcome is known truth at execution time, an opaque predication is T.

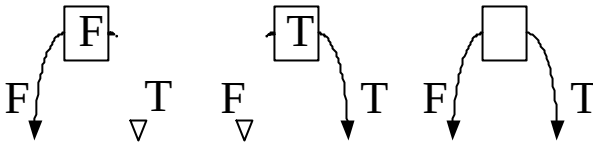


***Figure 2-1. The notion of different types of opaque predicates.***
***(Solid lines indicate paths that may sometimes be taken, dashed lines paths that will never be taken.)***

We design three kind methods of obfuscations to protect Java bytecode. Our new techniques are named as *destroying basic block obfuscation, replacing goto obfuscation* and *intersecting loop obfuscation*. We describe the three methods in the following sections.

### 2.1 Destroying Basic Block Obfuscation

A *basic block* is a sequence of instructions with single entry point and an exit point. Here we find five types of basic block in Java bytecode, as shown in Figure 2-2. If these basic blocks are destroyed, it will make decompiling unsuccessful.

- Load › Load › Consume_Two_NoComapre
- Compare › If
- Invoke › Invoke
- New › Dup
- Load › Load › Load › Array_Store

***Figure 2-2. Five types of basic blocks.***

*The destroying basic block obfuscation* must insert destroyed instructions in front of the last instruction of every basic block. For example, the basic block "Load, Load, Load, Array_Store", inserts destroyed instructions between the last "Load" and "Array_Store". The destroyed instructions are like conditional branch instructions and goto instructions. Figure 2-3 shows the *destroying basic block obfuscation*. The technique must take care of the goto instructions. The goto instructions must be located after the basic block and it is not in the basic block. Only in this way, the decompiler which using pattern matching technique will fail.

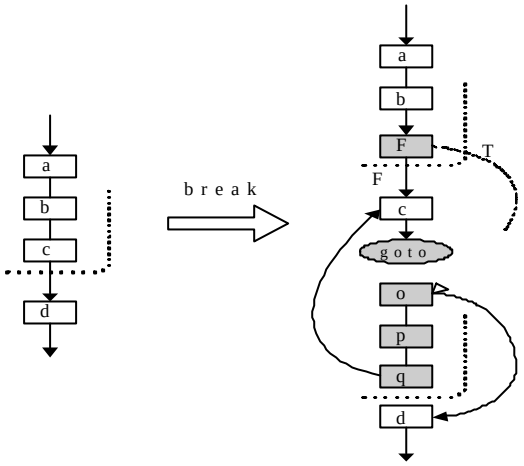


***Figure 2-3. The destroying basic block obfuscation. (Gray blocks are added additional instruction. F is an opaque predicate.).***

## 2.2 Replacing Goto Obfuscation

The Java language has no goto statement, but the Java bytecode instruction set does have a goto instruction. The *replacing goto obfuscation* is to replace goto instructions into conditional branch instructions. It makes control flow complicated and decompiler fails. Figure 2-4 is the obfuscated process of the *replacing goto obfuscation*.

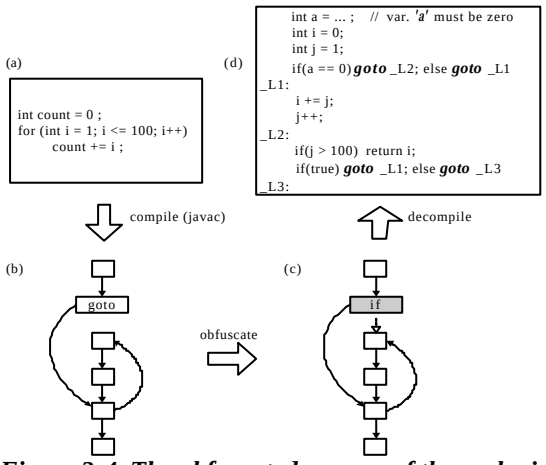


***Figure 2-4. The obfuscated process of the replacing goto obfuscation***

Figure 2-5 shows two simple obfuscation examples. Example 1 uses constant zero value to be a conditional branch, which always jump to Label1. But this kind of code would be removed out and returns to a goto instruction easily by an optimizer. So, example 2 is better. Of course, the local variable of example 2 must store zero value. In other words, a determined value of a conditional branch must not influence the original flow of a program segment; hence this is a fake conditional branch. However, this is a real conditional branch for a decompiler. The compiler just cannot distinguish its real function.

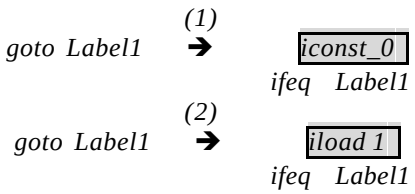


***Figure 2-5. Two examples of Replacing Goto Obfuscation.***

Table 2-1 is a program segment of calculating Boolean values. The goto instruction (pc=7) can be replaced into a conditional branch instruction in Table 2-1. If we didn't add compensated instructions, it will make stack state consistent, shown as in Table 2-2.

***Table 2-1. un-obfuscated program includes goto instruction.***

| before executing stack[top ...bottom] | PC | opcode | after executing stack[top... bottom] |
|---|---|---|---|
| [] | 0 | iload 2 | [int] |
| [int] | 3 | ifeq -> 10 | [] |
| [] | 6 | iconst_0 | [int] |
| [int] | 7 | goto -> 11 | [int] |
| [] | 10 | iconst_1 | [int] |
| [int] | 11 | istore 4 | [] |

In Table 2-1, after the goto instruction (pc=7) is executed, it has an integer item in stack. Before next instruction iconst_1 (pc=10) being executed, it has not any item in stack. The one before iconst_1 instruction (pc=10) is a goto instruction (pc=7); the two instruction's states of the stack are not related. So, the stack's state has no problem between goto and iconst_1 in Table 2-1.

But, if the goto instruction (pc=7) is replaced by a conditional branch instruction, it will make stack's state inconsistent. In Table 2-2, after ifeq condition branch instruction being executed (pc=9). It had two running paths of the program flow. One path was matched conditional value to jump istore4 (pc=13), that it produces the state of stack 1. The other path was not matched conditional value to execute the next instruction; it produces the state of stack 2. But these two stack's states are not consistent between ifeq (pc=9) and iconst_1 (pc=12).

***Table 2-2. The stack of the replacing goto obfuscation without compensated instructions.***

| before executing | | PC | opcode | after executing | |
|---|---|---|---|---|---|
| stack 1 | stack 2 | | | stack 1 | stack 2 |
| [] | [] | 0 | iload 2 | [int] | [int] |
| [int] | [int] | 3 | ifeq -> 12 | [] | [] |
| [] | [] | 6 | iconst_0 | [int] | [int] |
| [int] | [int] | 7 | iload 1 | [int, int] | [int, int] |
| [int, int] | [int, int] | 9 | ifeq -> 13 | [int] | [int] |
| [] | [int] | 12 | iconst_1 | [int] | [int, int] |
| [int] | [int, int] | 13 | istore 4 | [] | [int] |

So, when it occurs that the stack after executing a goto instruction is not consistent with the stack before executing next instruction, compensation instructions must be added to remain consistent for two stack's states.

***Table 2-3 Stack includes object reference.***

| before executing | PC | opcode | after executing |
|---|---|---|---|
| stack | | | stack |
| [] | 60 | aload_1 | [ref] |
| [ref] | 61 | iload_2 | [int, ref] |
| [int, ref] | 62 | iload_0 | [int, int, ref] |
| [int, int, ref] | 63 | ifeq -> 74 | [int, ref] |
| [int, ref] | 66 | aload 3 | [ref, int, ref] |
| [ref, int, ref] | 68 | invokevirtual 91 | [ref, int, ref] |
| [ref, int, ref] | 71 | goto -> 79 | [ref, int, ref] |
| [int, ref] | 74 | aload 3 | [ref, int, ref] |
| [ref, int, ref] | 76 | invokevirtual 92 | [ref, int, ref] |
| [ref, int, ref] | 79 | aastore | [] |

Table 2-3 is a part of the program. The most important part of instructions includes some object references in the stack. In Table 2-3 after executing the goto instruction (pc=74), the stack's content is "[ref, int, ref]". Before executing the next instruction "aload" (pc=74), the stack content is "[int, ref]". After obfuscating, it means that the compensation instruction for stack is to pop a '[ref]'. In practice, object pointers cannot decide whether they are the same or not in the bottom of two stacks. So, our obfuscation can only process goto instructions that don't have object pointers now.

In processing exception instructions there are some limits. The exception processor is executed only at exception time. The exception processor is not invoked in execution sequence instructions, and it cannot use a goto instruction to jump into the exception processor. Figure 2-6 shows using a goto instruction to skip the exception processor. It can avoid starting the exception processor in an original flow. If a goto instruction is used to skip the exception processor, this goto instruction cannot be replaced by a conditional branch instruction. This is for avoiding verifying failure by Java verifiers.

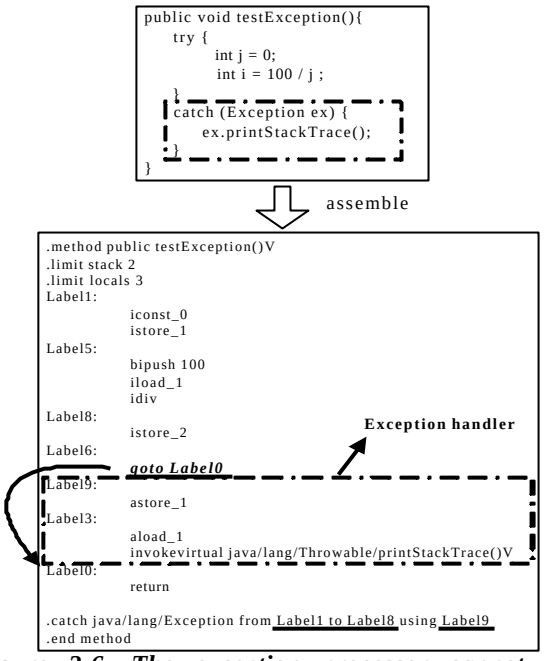


***Figure 2-6. The exception processor cannot be executed at execution sequence instructions.***

## 2.3 Intersecting Loop Obfuscation

Another method is by adding a control flow that Java high-level language cannot be present in program. It make decompiler to fail for Java [8].

The *intersecting loop obfuscation* uses two similar loops to intersect together (ex. for-loop), as Figure 2-7. These intersected loops are not permitted in any high-level language. Therefore, it can use to be an obfuscation technique.

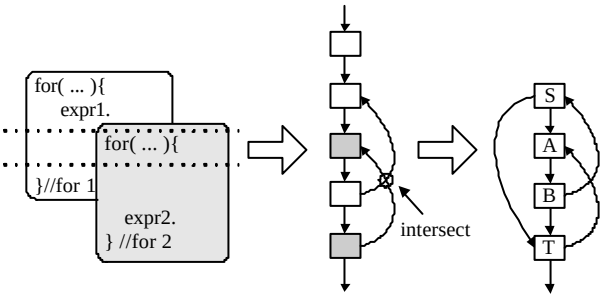


***Figure 2-7. The diagram of an intersecting loop.***

Strategies for avoiding verifying failure by the Java verifier, the stack states at every instruction must be consistent, and the stack states at entry point and left point also must keep consistent in the whole control flow. Without these conditions, the Java verifier will verify failure and terminate the program running. Beside, using a fake conditional branch to skip this intersected loop can avoid entering it in running time. And the performance will not be dropped off in the obfuscated program. Figure 2-8 shows it.

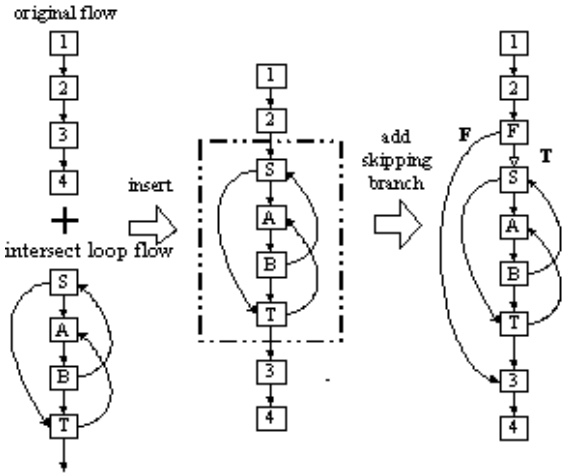


***Figure 2-8. The added process of the intersecting loop obfuscation.***

The above study in the *intersecting loop obfuscation* is called "single block", which inserted an intersect loop into two adjacent instructions. Although it can make decompilers fail, the obfuscated codes centralized will be easily to remove by attackers. We design a more complicated obfuscation technique, which we call it "multiple blocks" in the *intersecting loop obfuscation*.

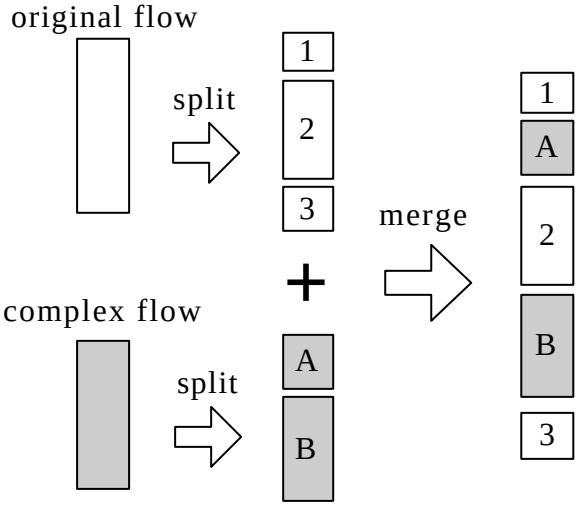


***Figure 2-9. The added process is the multiple blocks technique of the intersecting loop obfuscation.***

This multiple blocks technique divides obfuscated codes into several small blocks and breaks them into continuous blocks of the original program. The obfuscated codes are hard to remove out from the obfuscated program by attackers; hence hard to restore it to the original program. Figure 2-9 shows the added process of the multiple block technique. This is not a control flow diagram; it is a located diagram of blocks.

Figure 2-10 shows the detail of multiple blocks of the *intersect loop obfuscation*. It uses goto instructions to divide a control flow into several small blocks. The reason is that implementation is easy and it can force to jump. It can avoiding verify failure by the Java verifier.

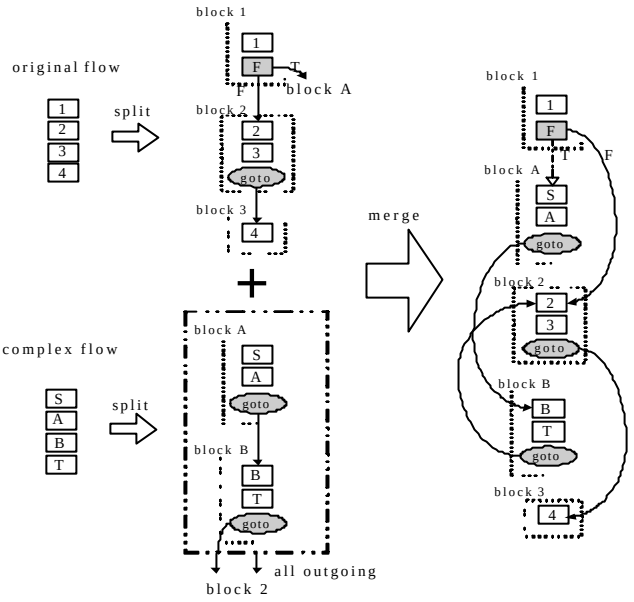


***Figure 2-10. The detail method of multiple blocks in the intersect loop obfuscation.***

In order to reduce the complication of Figure 2-10, we ignored pointers before splitting blocks. Actually, the pointers are existent. Except block 1 uses a conditional branch instruction, other blocks use goto instruction to jump to the next block for keeping the original flow. In the final step, every block merges together.

## 3. Result

### 3.1 Decompile test environment

The test environment of this research uses Windows 2000、256 MB DDR RAM and CPU AMD Athlon(tm) XP 1600+ 1.4 GHz. Tested decompiler environment is in Sun Java 2 SDK 1.3.1_05.

### 3.2 Decompile test result

For testing our designed obfuscation techniques in this paper, we collected decompile software from all over the world. Decompiles have 20 packages at least in the world. As Table 3-1 shows, we collected 16 packages of decompiler and tested obfuscations.

We used TicTacToe to be a target program. We decompiled an obfuscated TicTacToe program, which was obfuscated by our designed obfuscations, by different Java decompilers. In Table 3-1, the symbols were listed as follows:

- ×: It cannot produce a java file or a complete source code.
- △: It can produce a java file, but source code cannot be compiled.
- □ : After obfuscating, both decompilation and re-compilation were successful, but the program didn't execute correctly.
- ○ : After obfuscating, both decompilation and re-compilation were success, and the program executed correctly.

In Table 3-1, the "un-obfuscation" column listed results of decompiling an un-obfuscated program. The "destroy basic block" column listed test results of the d*estroying basic block obfuscation*. The "replace goto" column listed test results of the *replacing goto obfuscation,* and it shows all decompilers decompile failure. The "intersect loop" column listed test results of the *intersecting loop obfuscatio*n, and it shows all decompilers fail, too.

***Table 3-1. The test result of decompilation obfuscated program.***

| | Java Decompiler | Un-obfuscation | destroy basic block | replace goto | Intersect loop |
|---|---|---|---|---|---|
| 1 | Mocha | △ | × | × | × |
| 2 | SourceTec Decompiler | △ | × | × | × |
| 3 | Jad | ○ | △ | △ | △ |
| 4 | Front End Plus | ○ | △ | △ | △ |
| 5 | mDeJava | ○ | △ | △ | △ |
| 6 | Decafe Pro | ○ | △ | △ | △ |
| 7 | Cavaj Java Decompiler | ○ | △ | △ | △ |
| 8 | DJ Java Decompiler | ○ | △ | △ | △ |
| 9 | NMI's Java Class Viewer | ○ | △ | △ | △ |
| 10 | JReversePro | △ | × | × | × |
| 11 | JODE | ○ | × | △ | × |
| 12 | JCavaj Java Decompiler | ○ | × | △ | × |
| 13 | Dava Decompiler | ○ | ○ | × | × |
| 14 | Johnink | ○ | △ | △ | △ |
| 15 | ClassSpy | ○ | × | △ | △ |
| 16 | jAsrii | ○ | △ | △ | △ |

## 4. Conclusion

The task of making reverse engineering difficult is not easy. We design three new obfuscation techniques in this paper. We named them as *destroying basic block obfuscation*, *replacing goto obfuscation* and *intersecting loop obfuscation.* Our designed obfuscations are all control flow obfuscations.

Our obfuscation techniques are different from other obfuscations in published papers. The advantage of our obfuscation techniques is focused on attacking the weaknesses of decompilers. Our designed obfuscations effectively protect programs to be reverse engineering. As a result, both the *replacing goto obfuscation* and the *intersecting loop obfuscation* succeed to defeat all the decompilers.

## 5. Future work

The *replacing goto obfuscation* will improve the implementation techniques. Under the limit that Java verifier cannot verify failure, all goto instructions can be replaced by conditional branch instructions.

Besides, for protecting obfuscated programs from being de-obfuscated to original source code, we must improve further the patterns for complicated flows to insert the fake conditional branch instruction that is hard for decompilers to detect.

## Acknowledgements

The project is supported by NSC under project NSC (93-2213-E-006-105-).

## Reference

[1] Hans Peter van Vliet. "Mocha - The Java decompiler", http://wkweb4.cableinet.co.uk/jinja/mocha.html, January 1996.

[2] WingSoft Company. "JavaDis - The Java Decompiler", March 1997.

[3] Christian Collberg, Clark Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation -Tools for Software Protection", IEEE Transactions on Software Engineering vol.28, no.8, August 2002, pp.735-746

[4] Gleb Naumovicb, Nasir Memom, "Preventing Piracy, Reverse Engineering, and Tampering", IEEE Computer Society, July 2003, pp.64-71

[5] Hanpeter van Vliet, "Crema: the Java obfuscator", http://www.brouhaha.com/~eric/computers/mocha.html, 1996

[6] Christian Collberg, Clark Thomborson, Douglas Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Diego, California, United States, 1998, pp.184-196

[7] Douglas Low, "Java Control Flow Obfuscation", Master's Thesis, Department of Computer Science, University of Auckland, New Zealand, June 1998

[8] W. W. Peterson, T. Kasami, N. Tokura, "On the

capabilities of while, repeat, and exit instructions",
Communications of the ACM, Volume 16, Issue 8,
August 1973, pp.503-512