Three control flow obfuscation methods for Java software

T.W. Hou, H.Y. Chen and M.H. Tsai

Abstract: Three novel control computation (control flow) obfuscation methods are described for protecting Java class files. They are basic block fission obfuscation, intersecting loop obfuscation and replacing goto obfuscation. The basic block fission obfuscation splits some chosen basic block(s) into more basic blocks, in which opaque predicates and goto instructions are inserted to make decompiling unsuccessful. The intersecting loop obfuscation intersects two loops and then adds this pattern into programs. The intersecting loop structure should not appear in a Java source program or a class file. The replacing goto obfuscation replaces goto instructions with conditional branch instructions. The new methods were tested against 16 decompilers. The study also implemented multi-level exit obfuscation and single-level exit obfuscation for comparison. Both the intersecting loop obfuscation and the replacing goto obfuscation successfully defeated all the decompilers.

1 Introduction

Java compilers translate Java source code into '.class' files, which contain the Java bytecode for the classes. Much of the information about the source code is kept in the class files. Since the appearance of the first Java decompiler [1], the threat of reverse engineering has become worth noting. Without proper protection, class files can be easily decompiled and reverse-engineered into Java source code [2].

One approach against reverse engineering is obfuscation. Obfuscation is a process that keeps a program's functions but makes it difficult to decompile, rendering decompilers unable to derive usable source codes from class files. The program that performs obfuscating transformations automatically is called an obfuscator.

Obfuscation techniques are categorised as lexical obfuscations, data obfuscations, layout obfuscations and control obfuscations [3–5], as shown in Fig. 1.

Lexical obfuscations modify the lexical structure of the program, typically, scrambling identifiers. All the meaningful symbolic information of a Java program, such as classes, fields and method names, is replaced with meaningless information. An example of one such program is Crema [6], a Java obfuscator.

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

Layout obfuscations involve obscuring the logic inherent in a program. Examples are scrambling identifier names, removing comments and debugging information.

Control obfuscations make the control flow of a program difficult to understand. For example, the opaque predicate complicates the control flow by using conditional instructions that are always true (or false) [7]. The always true conditional instructions will branch to the original codes, whereas the false instruction will branch to codes arbitrarily inserted by the obfuscator. Control obfuscations are categorised as control aggregation obfuscations, control ordering obfuscations dispatcher obfuscation and control computation obfuscations, as illustrated in Fig. 1 and further explained in what follows.

Control aggregation obfuscations change the way in which instructions are grouped together. Inlining and outlining are two of the most effective ways by which methods and invocations of them can be obscured.

Control ordering obfuscations change the execution order of instructions, for instance loops can be set to sometimes iterate backwards instead of forwards.

Dispatcher obfuscations first flatten the control flow of a program. The structure of the flattened program becomes a dispatcher and some basic blocks. The dispatcher implements the control flow. Some NP-complete [8] or PSPACE-complete [9] methods are introduced in the dispatcher to cloak the program, which make the dispatcher difficult to trace.

Control computation obfuscations hide the real control flow of a program. One example is that instructions that have no effect can be inserted into a program. Control computation obfuscations are further categorised as smoke and mirrors obfuscations, high-level language breaking obfuscations and alter control flow obfuscations [3].

Smoke and mirrors obfuscations hide the real control flow behind instructions that are irrelevant, for example, by inserting dead codes into a program.

High-level language breaking obfuscations introduce features at the object code level for which there is no direct source code equivalent. For example, Java does not have a goto statement. Inserting goto instructions at the bytecode level could render decompilers unable to find suitable flow graphs.

Alter control flow obfuscations take a sequence of low-level instructions to construct an equivalent description at a higher level, thus removing abstractions from the program. For example, a for-loop in the C language source code can be transformed into an equivalent loop that uses 'if' and 'goto' statements.

© The Institution of Engineering and Technology 2006

IEE Proceedings online no. 20050010

doi:10.1049/ip-sen:20050010

Paper first received 4th March and in revised form 11th December 2005

The authors are with the Network Computing Laboratory, Department of Engineering Science, National Cheng Kung University, 1, Ta-Hsueh Road, Tainan, 701, Taiwan, Republic of China

E-mail: i14248@mail.hku.edu.tw

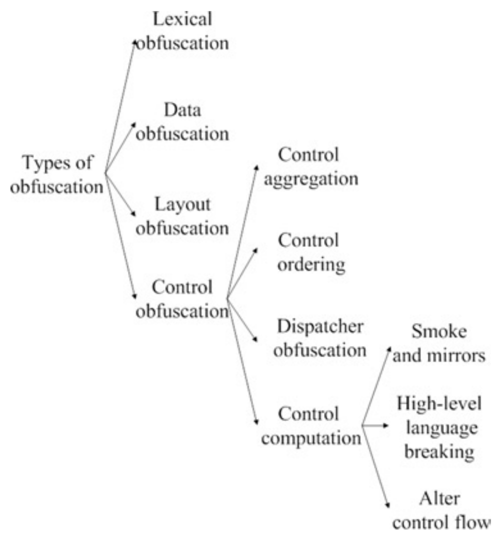


Fig. 1 *Categories of obfuscations*

Three new control computation (control flow) obfuscation techniques are proposed in this paper. They are named basic block fission obfuscation, intersecting loop obfuscation and replacing goto obfuscation. The basic block fission obfuscation destroys basic block(s) for which there are no direct source code equivalents. It is a high-level language breaking obfuscation. The intersecting loop obfuscation inserts a dead code that intersects two loops. It is a smoke and mirrors obfuscation. The replacing goto obfuscation replaces the bytecodes goto instructions with conditional instructions. It is an alter control flow obfuscation. The intersecting loop obfuscation is a kind of dead code guarded by an opaque predicate, where the predicate always directs control to flow in the original direction, bypassing the dead code. The basic block fission and replacing goto obfuscations are equivalent to the original flow and their codes are not bypassed.

We developed an automatic obfuscator, named 'crossover' [10], to implement the three obfuscations at the bytecode level. The challenge of developing the obfuscator was to pass Java Verifier, because inserting some bytecode instructions or changing control flows may make the stack inconsistent. We developed a simulator to find proper positions to insert obfuscation patterns safely.

Peterson *et al.* [11] proved that using if/loop/multi-level exit/node splitting techniques can convert any control flow into a well-formed program. They proposed two kinds of complex control flows, as shown in Fig. 2, where S, A, B, C, D, T are program nodes. Fig. 2*a* is an example of the multi-level exit control flow, and Fig. 2*b* is an example of the single-level exit control flow. These complex control flows cannot be translated into if–while programs without increasing their lengths and/or changing their execution sequences. They can be regarded as patterns to fail Java decompilers. They are implemented in our obfuscator to test the decompilers and for comparison to our approach.

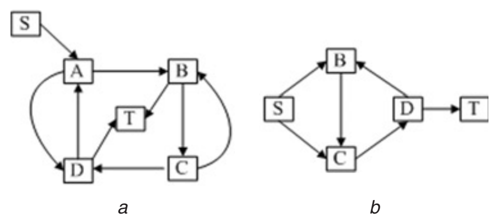


Fig. 2 *Two kinds of complex control flows [11]*
a Multi-level exit
b Single-level exit

2 Proposed obfuscation methods

Three new control flow obfuscations methods are proposed. All of them have been developed to find some structures in the class files and transform them into combinations of bytecode, which confuse the decompilers and then fail to generate the corresponding Java source code.

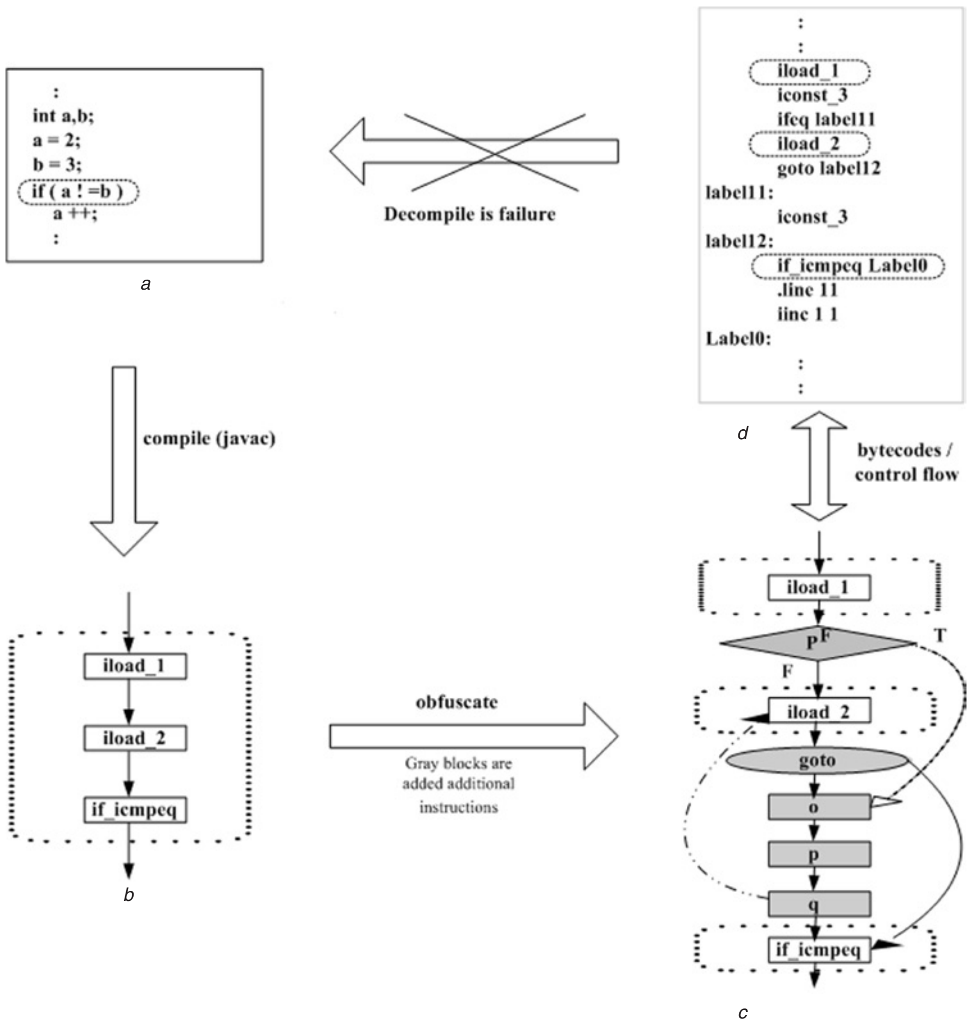
2.1 *Basic block fission obfuscation*

Some early decompilers used pattern matching to decompile programs. Pattern matching is to match instruction patterns to determine which program structure needs to be recreated [12]. A basic block is a block of instructions that starts with an entry point and finishes with an exit point. One kind of decompilers uses basic blocks to reverse programs to high-level language statements [13].

The basic block fission obfuscation method to counter such decompilers is to split some chosen basic block(s) into more basic blocks, where opaque predicates and goto instructions are inserted to make decompiling unsuccessful. An example of basic block fission obfuscation is shown in Fig. 3. A program segment is compiled into a basic block. After fission obfuscation, a few more basic blocks (grey) are generated. While the real control flow is kept, the 'grey' instructions are inserted to ensure the failure of decompilers. P^F is an opaque predicate made by using conditional instructions that always evaluate as false [7]. Some dummy blocks o, p and q are inserted for obscuring useful operations.

Java Verifier, the bytecode verifier of Java Virtual Machine, would find the obfuscated code unexecutable by checking a piece of code for type consistency and some other properties. For instance, the stack states of the entry point and exit point of each instruction must remain consistent throughout the whole control flow. It is not easy to arbitrarily add instructions into basic blocks. Five patterns of basic blocks are analysed to be suitable for the basic block fission obfuscation method, as summarised in Table 1. The patterns are composed of eight types of instructions, namely `Load`, `Consume_Two_NoMath`, `Compare`, `If`, `Invoke`, `New`, `Dup` and `Array_store`. The instruction types and their bytecode instructions are illustrated as follows.

- `Load`. This type of instructions does not consume an item and it put an item in the stack, such as `iload`, `aload`, `fload`, `dload`, `lload`, `bipush`, `sipush` and `ldc`.
- `Consume_Two_NoMath`. These instructions include `if_icmpeq`, `if_icmpge`, and `if_icmplt`. They consume two stack items, but exclude mathematic instructions, such as `iadd`, `iand` and `idiv`.
- `Compare`. The instructions are for comparison. They are `lcmp`, `fcmpl`, `fcmpg`, `dcmpl` and `dcmpg`.
- `If`. This type of instructions is for branch instructions, such as `ifeq`, `iflt`, `ifne` and `ifge`.
- `Invoke`. The instructions are to invoke methods, such as `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual`.
- `New`. It is the `new` instruction.
- `Dup`. It is the `dup` instruction.
- `Array_Store`. This type of instructions stores array items. They include `iastore`, `aastore`, `fastore`, `astore` and `lastore`.



```
:
int a,b;
a = 2;
b = 3;
if ( a ! =b )
  a ++;
:
```

```
        :
        :
        iload_1
        iconst_3
        ifeq label11
        iload_2
        goto label12
label11:
        iconst_3
label12:
        if_icmpeq Label0
        .line 11
        iinc 1 1
Label0:
        :
        :
```

Fig. 3 *Basic block fission obfuscation*

a Dotted block is a fission basic block pattern
b Fission basic block bytecodes belong to pattern 1 in Table 1
c Obfuscated program's control flow (grey blocks are added additional instructions)
d Fission basic block bytecodes

Table 1: Five types of basic block pattern that is suitable for the basic block fission obfuscation method and the bytecode instructions of each type of the pattern

Type	Basic block pattern
1	`Load, Load, Consume_Two_NoMath`
2	`Compare, If`
3	`Invoke, Invoke`
4	`New, Dup`
5	`Load, Load, Load, Array_Store`

Instruction types	Bytecode
`Load`	`iload, aload, fload, dload, lload, bipush, sipush, ldc`
`Consume_Two_NoMath`	`if_icmpeq, if_icmpne, if_icmpge, if_icmplt`
`Compare`	`lcmp, fcmpl, fcmpg, dcmpl, dcmpg`
`If`	`ifeq, iflt, ifne, ifge`
`Invoke`	`invokeinterface, invokespecial, invokestatic, invokevirtual`
`New`	`new`
`Dup`	`dup`
`Array_Store`	`iastore, aastore, fastore, astore, lastore`

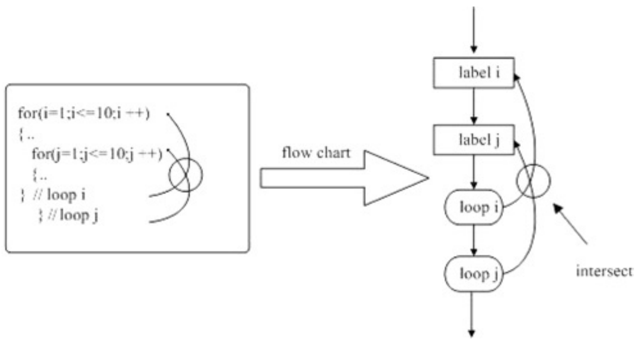


Fig. 4 *Diagram of intersecting loops*

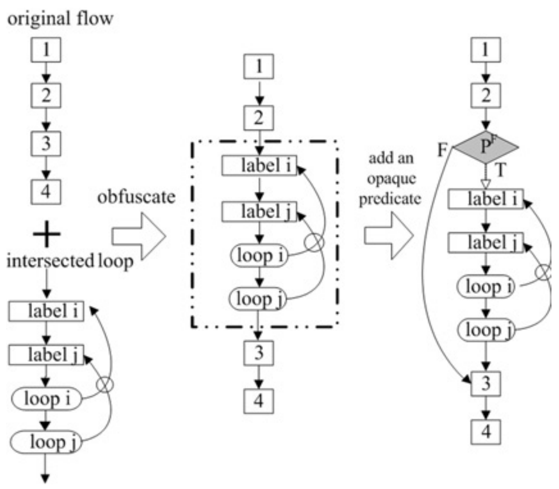


Fig. 5 *Processes of the intersecting loop obfuscation*

The basic block fission obfuscation method inserts some destroyed instructions in front of the last instruction of every target basic block. For example, for the basic block of type '`Load`, `Load`, `Consume_Two_NoMath`', obfuscation instructions are inserted between the last '`Load`' and '`Consume_Two_NoMath`'. The inserted instructions are opaque predicates and goto instructions. By using this technique, the pattern matching decompilers may fail to reconstruct the original statement.

2.2 *Intersecting loop obfuscation*

A decompiler would fail to reconstruct Java statements because of a reasonless control flow [14]. For instance, to add a control flow that should not have appeared in a Java source program. Or a professional Java compiler would never generate such a structure. The intersecting loop obfuscation method intersects two loops and then adds this pattern into a program, as shown in Fig. 4.

Fig. 5 shows the process of adding intersecting loops into the original flow. A false opaque predicate is introduced to skip this intersected loop, which avoids the program from entering the intersected loop block during run time. It is also vital to avoid the verification failure caused by Java Verifier.

In Fig. 5, although this could make the decompiler fail, the obfuscated codes are put together in such a way that they may easily be detected and removed. One solution is to manufacture some cheap, resilient and stealthy opaque constructs that can be applied to prevent the pattern-matching attack, as suggested by Collberg *et al.* [7]. However, we introduce another obfuscation technique, called multiple-block intersecting loop obfuscation. As shown in Fig. 6, both original flows and the intersected-loop

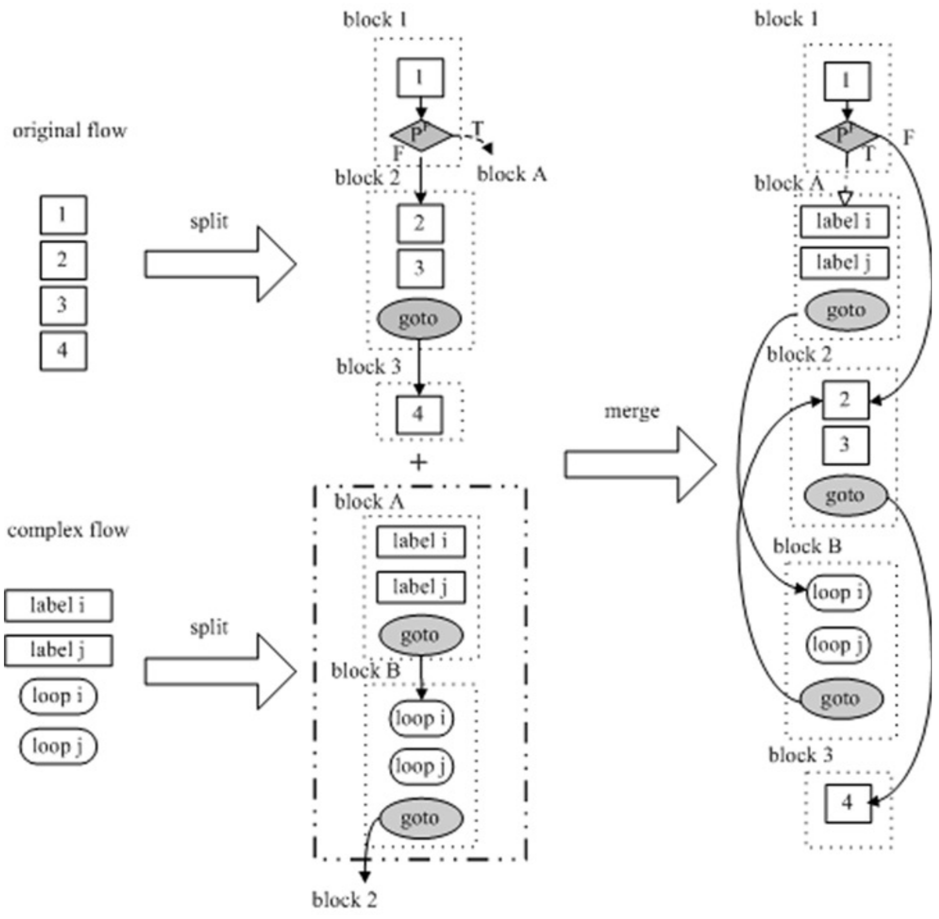


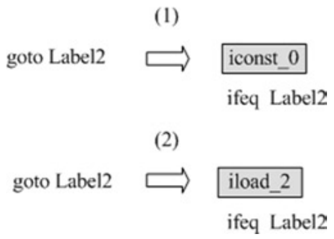
Fig. 6 *Transformation process of multiple-block in intersecting loop obfuscation*

block are split first. Some goto and opaque predicate instructions are inserted to complicate the program flow. Then they are merged to form a more complex structure, on which pattern matching is not applicable.

2.3 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 method replaces goto instructions with conditional branch instructions. Fig. 7 shows an example of the replacing goto obfuscation technique. The original code segment is shown in Fig. 7*a*. Its flow graph and bytecodes are shown in Fig. 7*b*. After obfuscation, the new flow and bytecodes are shown in Fig. 7*c*.

The new conditional branch instruction must not influence the original flow of a program segment; hence it is a fake conditional branch. However, this is seen as a real conditional branch for a decompiler. A decompiler would not distinguish its real function. There are two alternatives to implement the always-predict-false methods, as shown in Fig. 8. The first one uses constant zero (false) for a conditional branch, which always jumps to Label 2. But this kind of code could easily be moved out and replaced by a goto instruction by an optimiser, so the second is better. Of course, the local variable must be zero.



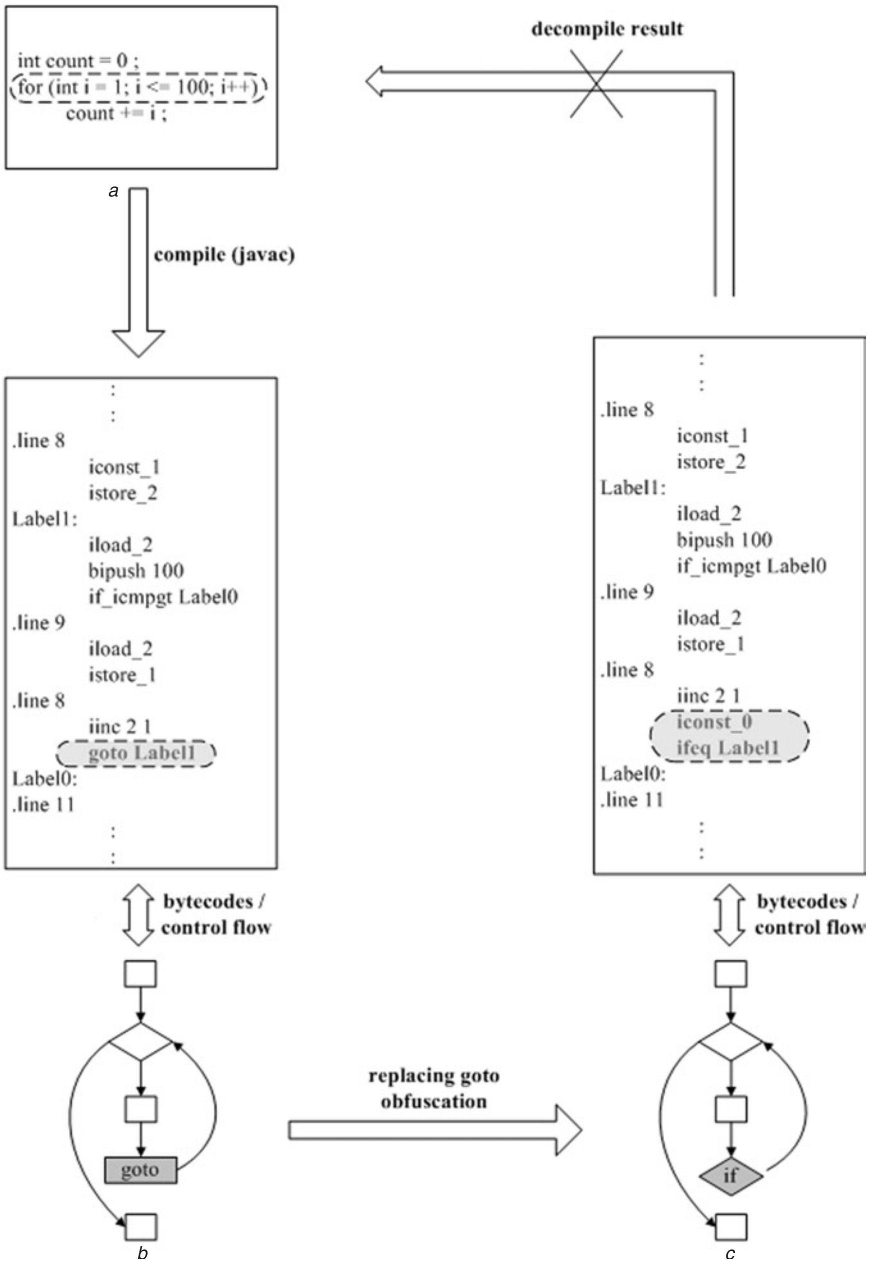
```
goto Label2   ⇨   iconst_0
                  ifeq Label2

goto Label2   ⇨   iload_2
                  ifeq Label2
```

Fig. 8 *Two alternatives of the replacing goto obfuscation method*

3 Testing results

The testing environment was a PC-clone with Sun Java 2 SDK 1.3.1_05, Windows 2000, 256 MB DDR RAM and



```
int count = 0 ;
for (int i = 1; i <= 100; i++)
    count += i ;
```

```
        :
        :
.line 8
        iconst_1
        istore_2
Label1:
        iload_2
        bipush 100
        if_icmpgt Label0
.line 9
        iload_2
        istore_1
.line 8
        iinc 2 1
        goto Label1
Label0:
.line 11
        :
        :
```

```
        :
        :
.line 8
        iconst_1
        istore_2
Label1:
        iload_2
        bipush 100
        if_icmpgt Label0
.line 9
        iload_2
        istore_1
.line 8
        iinc 2 1
        iconst_0
        ifeq Label1
Label0:
.line 11
        :
        :
```

Fig. 7 *Process of replacing goto obfuscation*

Table 2: Test results for the decompilation of obfuscated programs

Java decompiler	No obfuscation	Basic block fission	Replacing goto	Intersecting loop	Multi-level exit	Single-level exit
Cavaj Java decompiler v1.11 [15]	○	△	△	△	△	△
ClassSpy v2.0 [16]	○	×	△	△	△	△
Dava decompiler v1.0.0 [17]	○	○	×	×	○	○
Decafe Pro v3.6 [18]	○	△	△	△	△	△
DJ Java decompiler v3.9.9.91 [19]	○	△	△	△	△	△
Front end plus v1.00 [20]	○	△	△	△	△	△
Jad v1.5.8e2 [21]	○	△	△	△	△	△
jAscii v1.0.20 [22]	○	△	△	△	△	△
JCavaj Java decompiler v1.00 [23]	○	×	△	×	△	×
JODE v1.1.1 [24]	○	×	△	×	△	×
JReversePro v1.4.1 [25]	△	×	×	×	×	×
Jshrink v2.36 [26]	○	△	△	△	△	△
mDeJava v1.0b [27]	○	△	△	△	△	△
Mocha v1.0b [1]	△	×	×	×	×	×
NMI's Java class viewer v4.7 [28]	○	△	△	△	△	△
SourceTec decompiler v1.1 [29]	△	×	×	×	×	×

Cross denotes that the decompiler could not produce a Java file or a complete source code; open triangle denotes that the decompiler could produce a Java file, but the generated source code had syntax errors; open square (none in this category) denotes that after obfuscating, both decompilation and re-compilation were successful, but the program did not execute correctly; open circle denotes that after obfuscating, both decompilation and re-compilation were successful, and the program executed correctly

AMD Athlon(tm) XP 1600+ 1.4 GHz CPU. TicTacToe served as the target program. The bytecodes of TicTacToe were obfuscated separately by the proposed methods. Then the obfuscated TicTacToe program was fed into 16 separate available Java decompilers [1, 15–29].

Table 2 summarises the results. Fourteen decompilers could rebuild the original source code if no obfuscation was applied. Only Dava decompiled the basic block fission obfuscated code. Both the replacing goto obfuscation and the intersecting loop obfuscation succeeded in defeating all the decompilers, providing to be more effective than the multi-level exit obfuscation and the single-level exit obfuscation methods.

4 Conclusion

The task of making reverse engineering more difficult is evolving. Three new control computation (control flow) obfuscation techniques are devised, implemented and tested with 16 available decompilers. They are named basic block fission obfuscation, replacing goto obfuscation and intersecting loop obfuscation. The proposed obfuscations effectively protect programs from reverse engineering, and both the replacing goto obfuscation and the intersecting loop obfuscation succeeded in defeating all the decompilers.

The obfuscated program must keep its original flow, so some opaque predicate instructions are needed, but they cannot be added at any place because of the Java Verifier. More techniques need to be developed to ensure that they can stay ahead of the next generation of decompilers.

5 Acknowledgments

This project was partially supported by NSC under project NSC 93-2213-E-006-105. The authors wish to thank editors and the anonymous referees for their suggestions and comments.

6 References

1 van Vliet, H.P.: 'Mocha – The Java decompiler', http://www.brouhaha.com/~eric/software/mocha/, v1.0b, January 1996

2 WingSoft Company: 'JavaDis – The Java Decompiler', http://www.wingsoft.com/wingdis.html, March 1997

3 Low, D.: 'Java control flow obfuscation'. Master's Thesis, Department of Computer Science, University of Auckland, New Zealand, June 1998

4 Collberg, C., and Thomborson, C.: 'Watermarking, tamper-proofing, and obfuscation – Tools for Software Protection', *IEEE Trans. Softw. Eng.*, 2002, **28**, (8), pp. 735–746

5 Naumovich, G., and Memon, N.: 'Preventing piracy, reverse engineering, and tampering', *Computer*, 2003, **36**, (7), pp. 64–71

6 van Vliet, H.P.: 'Crema: the Java obfuscator', http://www.brouhaha.com/~eric/computers/mocha.html, 1996

7 Collberg, C., Thomborson, C., and Low, D.: 'Manufacturing cheap, resilient, and stealthy opaque constructs'. Proc. 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, San Diego, CA, USA, 1998, pp. 184–196

8 Wang, C.: 'A security architecture for survivable systems', PhD Dissertation, Department of Computer Science, University of Virginia, ftp://ftp.cs.virginia.edu/pub/dissertations/2001-01_abs.html, 2001

9 Chow, S., Gu, Y., Johnson, H., and Zakharov, V.A.: 'An approach to the obfuscation of control-flow of sequential computer programs'. Proc. Information Security Conf., ISC 2001, *Lect. Notes Comput. Sci.*, **2200**, pp. 144–155

10 Hou, T.W., Chen, H.Y., and Tsai, M.H.: 'Crossover obfuscator', http://www.nc.es.ncku.edu.tw/crossover/, November 2005

11 Peterson, W.W., Kasami, T., and Tokura, N.: 'On the capabilities of while, repeat, and exit instructions', *Commun. ACM*, 1973, **16**, (8), pp. 503–512

12 Miecznikowski, J., and Hendren, L.: 'Decompiling Java using staged encapsulation'. Proc. 8th Working Conf. on Reverse Engineering (WCRE'01), Stuttgart, Germany, 2–5 October 2001, pp. 368–374

13 Proebsting, T.A., and Watterson, S.A.: 'Krakatoa: decompilation in Java (Does bytecode reveal source?)'. Proc. 3rd USENIX Conf. on Object-Oriented Technologies and Systems, Portland, OR, USA, June 1997

14 Lam, P.: 'Of graphs and Coffi grounds: decompiling Java', Sable Technical Report no. 6, McGill University School of Computer Science, Sable Research Group, September 1998

15 Sureshot Software Co., Ltd: 'Cavaj – The Java decompiler', http://www.bysoft.se/sureshot/cavaj/, v1.11, June 2002

16 Gonsalves, M.A.: 'ClassSpy – The Java decompiler', v2.0, http://www.brothersoft.com/Software_Developer_Miscellaneous_ClassSpy_6712.html, November 2002

17 Miecznikowski, J., and Hendren, L.: 'Dava – The Java decompiler', http://www.program-transformation.org/Transform/DecompilationDava#About_Dava, v1.0.0, 2001

18 Decafe: 'Decafe – The Java decompiler', http://descargas.terra.es/informacion_extendida.phtml?n_id=8685&plat=1, v3.6, 1999)

19 Neshkov, A.: 'Welcome to DJ Java decompiler', http://members.fortunecity.com/neshkov/dj.html, v3.9.9.91, 2002

20 Cowley, M.: 'FrontEnd Plus – The Java decompiler', http://www.softpile.com/Development/Java/Review_03171_index.html, v1.00, March 2001

21 Kouznetsov, P.: 'Jad – The Java decompiler', http://www.kpdus.com/jad.html, v1.5.8e2, 2001

22 D&C Software Solutions: 'jAscii – The Java decompiler', http://www.program-transformation.org/Transform/DecompilationJasciiTest, v1.0.20, September 2003

23 Sureshot Software Co., Ltd: 'JCavaj – The Java decompiler', http://www.sureshotsoftware.com/jcavaj/manual.html#chapter1, v1.00, 2002

24 Hoenicke, J.: Canonic: 'JODE – The Java decompiler', http://jode.sourceforge.net/, v1.1.1, May 2001

25 GNU GPL: 'JReversePro – The Java decompiler', http://jrevpro.sourceforge.net/, v1.4.1, February 2002

26 Eastridge Technology: 'Jshrink – The Java decompiler', http://www.e-t.com/jshrink.html, v2.36, 1997

27 MoleSoftware: 'mDeJava – The Java decompiler', http://molesoftware.hypermart.net/, v1.0b, August 2000

28 Lemaire, B.: 'NMI's Java Class Viewer – The Java decompiler', http://www.jreveal.org/cgi-bin/resource.pl?resid=4397, v4.7, February 1999

29 SourceTec Software Co., Ltd: 'SourceTec – The Java decompiler', http://www.sothink.com/product/javadecompiler/index.html, v1.1, 1997

Copyright of IEE Proceedings -- Software is the property of Institution of Engineering & Technology and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.