A Java Program Tamper-proofing method

Xuesong Zhang, Fengling He, Wanli Zuo
College of Computer Science and Technology, Jilin University
Changchun, 130012, P.R.China
xs_zhang@126.com, Hefl@jlu.edu.cn, wanli@jlu.edu.cn

Abstract

Illegal distribution and use of software is a big ad hoc problem in software industry today, especially on the growing market of Java software product. Tamper-proofing techniques will disable some or all of the program's functionality once they detect any unwanted modifications during run time. This paper presents the attempt to Java software tamper-proofing technology by dynamic monitoring. The Java method's control flow graph is described by regular expressions, which can be regarded as a language. Certain stack signature in these control flows is also computed. During program execution, the protected method's control flow and stack signature information is verified at runtime by a monitor thread which checking whether the execution trace is accepted by the language, and the signature belongs to the right method. Experimental result shows that this tamper-proofing method is especially suitable for non-computation dense Java applications.

1. Introduction

With the development of distributed computing technique, a lot of new computing models have emerged, such as grid, mobile agents and peer-to-peer etc. As a neutral language, Java has been widely applied in these areas. In order to meet this kind of platform independent characteristic, Java introduces a form of symbolic link technique, witch stores all the type and interface information into class file's constant pool. These information provide the loading on demand and mobile property, but also facilitate decompilation at the same time.

Faced with Java bytecode, traditional software encryption technology gains little effect. Till now, the main technique used in protecting Java programs is software obfuscation. Several researchers have published papers on software obfuscation using automated tools and code transformations [1, 2, 3, 4, 5]. One idea is to use language-based tools to transform a program (most easily from source code) to a functionally equivalent program which presents greater reverse engineering barriers. If implemented in the form of a pre-compiler, the usual portability issues can be addressed by the back-end of standard compilers. Collberg [6] contains a wealth of additional information on software obfuscation, including notes on: a proposed classification of code transformations; the use of opaque predicates for control flow transformations; initial ideas on metrics for code transformations; program slicing tools; and the use of (de)aggregation of flow control or data.

The tamper-proofing techniques [7, 8, 9] in general are designed to detect or sense any type of tampering of a program. Once such tampering is detected, one of many possible actions could be taken by the anti-tamper part of the software. These actions could include disabling the software, deleting the software, or making the software generate invalid results rendering it useless to the tampering adversary. However, because Java bytecode cannot access stack directly, and does not able to modify its own code dynamically, a lot of traditional software tamper-proofing technique cannot be directly applied to Java programs. Further more, to protect the Java class files from modifications, the inserted (protective) code must pass the JVM bytecode verifier, which checks the format, the environment, and whether context of Java class files is correct or not.

This paper proposes one Java program tamper-proofing methodology based on thread monitoring. The monitoring thread must have some knowledge of the monitored program's meta-structure, some notion of program semantics with respect to the tamper protected domain. This domain in our work is control flow

The work of Xuesong Zhang, Fengling He and Wanli Zuo were supported in part by the Research of Basic Application of China under grants NO. 20070533

978-0-7695-3486-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SecTech.2008.28

integrity and stack signature (Figure 1). During program execution, each protected method will send their control flow (execution path) and stack information to the monitor thread, the monitor thread checks these information to ensure no tampering occurred. Otherwise, some action will be taken such as stop the program. We believe that all tampering methods, be they data tampering or memory tampering, eventually exhibit themselves in the control flow or the stack signature corruption. Hence, all types of tampering can be captured if only incorporated into an anti-tamper system design paradigm.

The rest of this paper is organized as follows. In Section 2, a Java program tamper-proofing model is

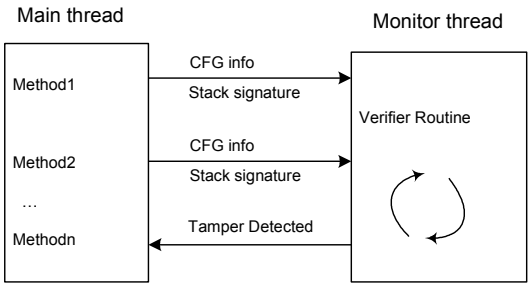

Figure 1 Protecting model overview

proposed. Section 3 gives the experimental result. Finally, Section 4 concludes the paper with some future work.

2. Tamper-proofing model

2.1. Control flow graph

A control flow graph (CFG) can be viewed as a Directed Acyclic Graph (DAG) of a program's control flow consisting of basic blocks as nodes, each of which are terminated by a control transfer instruction, such as a jump or branch, such as goto or ifeq in Java bytecode. Each basic block has at most two successors, and at least one predecessor (except for the entry block for a program which has no predecessor).

Figure 2 shows a Java method that that computes Fibonacci numbers. Figure 3 gives the corresponding method's CFG. From which we can see that block 1 and block 5 are control blocks, whose associated instruction is able to modify the program flow. The other blocks are sequential blocks.

```
int fib(short m){
  int f0 = 0, f1 = 1, f2 = 0;
  if(m <= 1)
    return m;
  else{
    for(int i = 2; i <=m; i++){
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }
    return f2;
  }
}
```

Figure 2. A Java Fibonacci method

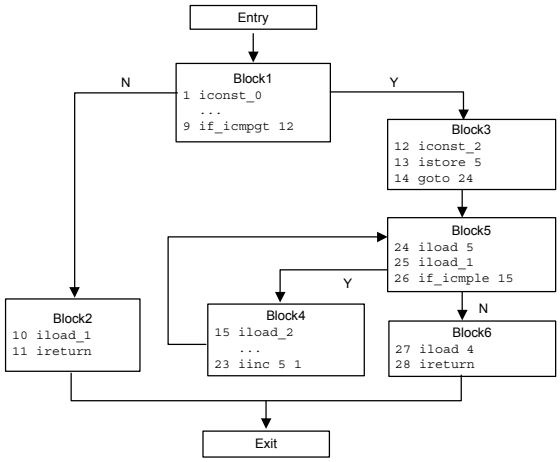

Figure 3. CFG of Fibonacci method

2.2. Flow Signature using Regular Expression

In the proposed approach the problem of control-flow checking is tackled resorting to a new signature approach able to identify all the allowed program flows executions. These program flows executions are identified by all the possible paths in the related Labeled CFG starting from the " Entry" block and ending in the "Exit" block (Figure 4). The defined signature is stored and checked at run time.

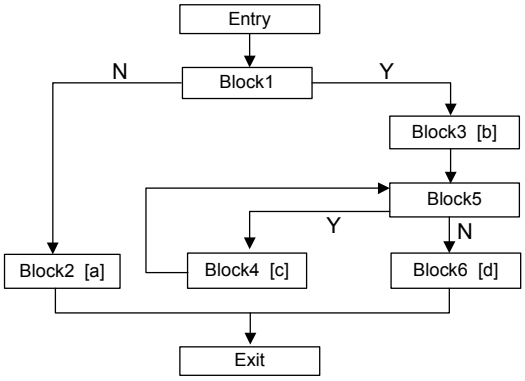

Figure 4. Labeled control flow graph

The novelty of our approach is in the use of regular

expressions to calculate the control flow signature, instead of classical approaches based on hashing functions. Each branch-free block in the CFG is labeled with a unique symbol named block symbol. Using this notation the allowed program flows executions generate a language composed by all the strings obtained by the concatenation of the block symbols composing the correspondent path in the Labeled CFG. This set of strings can be considered as a language L in the sense of compiler theory. This language can be represented in a formal way as:

$L = (\Sigma, R)$

Where:

- Σ is the input alphabet composed by all the defined block symbols;
- R is a regular expression able to generate all the necessary strings.

The language L can represent a signature for the target application program.

A program flow execution is allowed only if the corresponding path in the Labeled CFG generates a string (Control String) belonging to the language L, in the other case a flow error has occurred. Considering the example of Figure 3 the language L is:

$L = (\Sigma, R)$

Where

- Σ = (a, b, c, d)
- R = a | b (c)* d

If an execution produce the control string S = "bcccd" we can say that the execution belongs to the space of correct executions, whereas in case of the control string S="abccd" a program tamper must has occurred since the string is not recognized by the language L.

The choice of using regular expressions for signature computation relies on the high expressivities of this formalism and the high simplicity in storing and manipulating this kind of strings. Using this formalism, the problem of control flow monitoring is reduced to the problem of generating the control strings during the execution of the application program and verifying for their correctness by checking whether it is accepted by the language L.

2.3. Stack signature using constant value

All Java instructions are stack-based. Stack is the most important site for exchanging data. There are many instructions in Java bytecode that push a constant onto stack such as iconst, lconst, bipush, and ldc etc. When a Java file is compiled into class files, these instructions may be distributed in the method's code. Due to their constant feature, these instructions can be accumulated to form a stack trace — signature. However, since the CFG of a Java program method contains many control blocks, the execution path varies on different condition, which means we could not get a fixed signature for a method. The conditional branch is classified into the following two cases:

- Selection: This kind of branch corresponds to the if-else statement in Java language. The accumulated value is calculated for each and every branch. All signatures a method can have would be 2^n, where n is the selection branch numbers. Facing with a complicated method, signature numbers will get very large, and need more space to store. Currently, we only generate the first five selection branch's signature for complicated method.
- Loop: This kind of branch corresponds to the for, while statement in Java language. If the initial and loop condition all are const, we generate signature for the loop body, otherwise, the execution time is indefinite, we cannot get a constant signature.

Figure 5 shows the bytecode added to the program. We maintain a table for all protected methods, contains both the stack signature and the CFG regular expression. After each method execution, a check is performed to find any tampering.

3. Experimental result

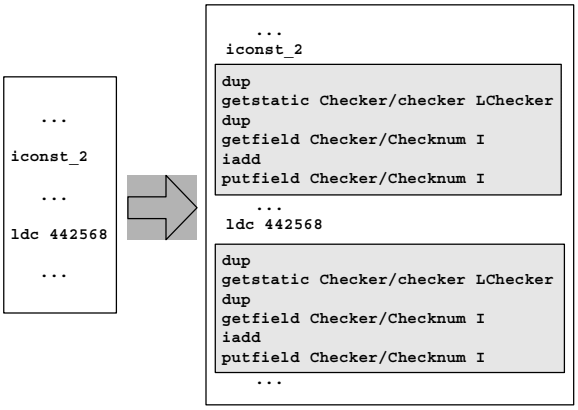

Figure 5. Stack signature calculation

We extend sandmark, and apply our scheme to five java programs. Currently, only relatively small method (branch numbers less than 20) is protected, method with exception tables are also omitted, which will lead to very complicated control flow analysis at bytecode level. Some of the programs are Java applets which will never terminate without user action. We only

measure their execution time in finite cycles. For an example, the ASM program will simulate the stock market forever, and the corresponding execution time given in Table 1 is based on the first thirty cycles.

From Table 1, the program size increasing ratio after protected lies in between 1.03 and 1.20. The average execution time is more than ten times slower than the original programs. Communications and synchronizations between protected methods and the monitor thread form a bottleneck to program efficiency. However, these test programs are dense calculation based tasks, which include a lot of loop bodies. For an ordinary graphical user interface Java program, most methods are executed sequentially, and few of then contain loop body, the reaction delay we have tested is very small.

Table 1. Experimental result

Program	Description	Method Protected		Before protection	After protection	Ratio
Imageview	Image file viewer	12	Jar file size (byte)	8237	9845	1.20
			Execution time (sec)	12.71	183.45	14.43
MultiSort	Collection of fifteen sorting algorithms	25	Jar file size (byte)	14558	16472	1.13
			Execution time (sec)	105.5	1207.9	12.04
Draw	Draw random graphs	23	Jar file size (byte)	16772	18011	1.07
			Execution time (sec)	6.03	109.23	18.11
ASM	Artificial stock market	35	Jar file size (byte)	87796	90125	1.03
			Execution time (sec)	31.20	452.66	14.50
DataReport	Report generator for electric power data	42	Jar file size (byte)	59030	68555	1.17
			Execution time (sec)	8.84	212.37	24.02

4. Conclusion

In this paper we introduced a tamper-proofing method by partition the Java program into two parts—the main part which does the original job, the monitor part which perform method's control flow and stack signature integrity checking. If the protected program is decompiled to Java source code successfully, the resilience of the protection mechanism would decrease greatly. There are two solutions to deal with this situation: One is resorting to the obfuscation techniques such as [10] which can be used to prevent the protected programs from being decompiled. The other one is to coupling the main part and the monitor part, make then both do parts of ordinary program job and monitoring each other at the same time.

In the next step, we optimize the proposed method to improve its efficiency, and perform variety kind of attacks to evaluate its true effect.

References

[1] Linn, C., Debray, S., Obfuscation of executable code to improve resistance to static disassembly. In: *ACM Conference on Computer and Communications Security*. 2003

[2] C. Wang et al., Software Tamper Resistance: Obstructing Static Analysis of Programs, tech. report CS-2000-12, Dept. Computer Science, Univ. Virginia, Charlottesville, 2000.

[3] Michael Batchelder and Laurie Hendren, Obfuscating Java: The Most Pain for the Least Gain. CC 2007, LNCS 4420, pp. 96–110, 2007.

[4] C. Collberg, C. Thomborson, D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", *Proc. Symp. Principles of Programming Languages* (POPL'98), Jan. 1998.

[5] C. Collberg, C. Thomborson, D. Low, "Breaking Abstractions and Unstructuring Data Structures", *IEEE International Conf. Computer Languages* (ICCL'98), May 1998.

[6] C. Collberg, C. Thomborson, D. Low, "A Taxonomy of Obfuscating Transformations", Technical Report 148, Dept. Computer Science, University of Auckland (July 1997).

[7] AUCSMITH, D.. Tamper-resistant software: an implementation. In *Proceedings of the 1st International Workshop on Information Hiding*, LNCS 1174, London, UK, Springer-Verlag, 317-333, 1996.

[8] LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D.,MITCHELL, J. and HOROWITZ, M., Architectural support for copy and tamper resistant software. In *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOSIX), November 2000, 169-177.

[9] LIE, D., MITCHELL, J., THEKKATH, C.A. and HOROWITZ, M., Specifying and verifying hardware for tamper-resistant software. *In Proceedings of IEEE Symposium on Security and Privacy*, Berkeley, CA., May 11 - 14, 2003, 166.

[10] Batchelder, M., and Hendren, L. J. Obfuscating java: The most pain for the least gain. In *Proceedings of the International Conference on Compiler Construction*. Springer, 2007, v. 4420 of *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pp. 96-110.