

# A Framework for Obfuscated Interpretation

†‡Akito Monden †Antoine Monsifrot †Clark Thomborson

†Department of Computer Science  
The University of Auckland  
Private Bag 92019, Auckland, New Zealand

{antoine, cthombor}@cs.auckland.ac.nz

‡Graduate School of Information Science  
Nara Institute of Science and Technology  
8916-5 Takayama, Ikoma, Nara 630-0101, Japan

akito-m@is.aist-nara.ac.jp

## Abstract

Software protection via obscurity is now considered fundamental for securing software systems. This paper proposes a framework for obfuscating the program interpretation instead of obfuscating the program itself. The obfuscated interpretation enables us to hide functionality of a given program  $P$  unless the interpretation being taken is revealed. The proposed framework employs a finite state machine (FSM) based interpreter to give the context-dependent semantics to each instruction in  $P$ ; thus, attempts to statically analyze the relation between instructions and their semantics will not succeed. Considering that the instruction stream (execution sequence) of  $P$  varies according to the input to  $P$ , we give a systematic method to construct  $P$  whose instruction stream is always interpreted correctly regardless of its input. Our framework is easily applied to conventional computer systems by adding a FSM unit to virtual machines such as Java Virtual Machine (JVM) and Common Language Runtime (CLR).

*Keywords:* Software Protection, Obfuscation, Encryption

## 1 Introduction

Many software systems rely on the obscurity of their implementation, to increase the security against hostile end-users. For example, in typical software digital rights management (DRM) system, its implementation must be obscure, i.e. must not be understood by end-users, since it contains cryptographic keys and algorithms that need to be kept secret (Chow et al. 2002). Such obscurity is also needed in embedded software of consumer electric devices, e.g. mobile phones and set-top boxes, since they are also sensitive to attacks by hostile users (The U.K. Parliament 2002). In addition, it is often difficult to prohibit physical access to the software implementation since even embedded software requires being easily updated.

In order to hide secrets in software implementation, software obfuscation techniques have been proposed (Cohen 1993, Collberg and Thomborson 2002, Kanzaki et al. 2003, Sander and Tschudin 1998). Software obfuscations transform a program so that it is more

difficult to understand, yet is functionally equivalent to the original program. However, there is no evidence those techniques are powerful enough to hide secrets in a program (Barak et al. 2001). Given enough time and effort, the obfuscated program can be understood by hostile users since it still contains all the necessary information to be thoroughly understood. Although software obfuscations are practically useful to some extent, a variety of complementary techniques are needed to dissuade the widest possible range of attackers.

Instead of obfuscating the program itself, this paper gives an idea for obfuscating the program interpretation. If the interpretation being taken is obscure and thus it can not be understood by a hostile user, the program being interpreted is also kept obscure since the user lacks the information about “how to read it.” This idea is similar to the randomized instruction-set approach (Barrantes et al. 2003); however, in the randomization approach, the interpretation itself is not obscure because randomized instructions still have one-to-one map to their semantics, although the map can be occasionally changed (Kc, Keromytis, and Prevelakis 2003). On the other hand, our aim is to give a dynamic map between instructions and their semantics.

In this paper we propose a framework for constructing an interpreter  $W$ , which carries out *obfuscated interpretations* for a given program  $P$ , where  $P$  is a translated version of an original program  $P_0$  written in a common (low level) programming language (such as Java bytecode and x86 assembly.) The obfuscated interpretation means that an interpretation for a given instruction  $c$  is not fixed; specifically, the interpretation for  $c$  is determined not only by  $c$  itself but also by previous instructions input to  $W$  (Figure 1).

In order to realize the obfuscated interpretation in  $W$ , we employ a FSM that takes as input an instruction  $c$  where each state makes a different interpretation for  $c$ . Since transitions between states are made according to the input, the interpretation for a particular type of instruction varies

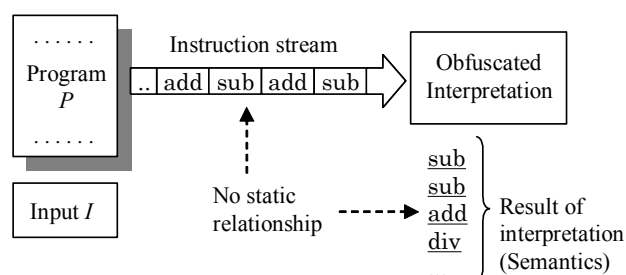


Figure 1. Concept of obfuscated interpretation

with respect to previous inputs. Such  $W$  we call a FSM-based interpreter. In our framework,  $W$  is built independent of  $P_0$ ; thus, many programs run on a single interpreter  $W$ , and any of the programs can be easily replaced to a new program for the sake of updating.

In some sense, the mechanism of obfuscated interpretation is a kind of stream cipher where a ciphered bit sequence is decoded one bit at a time dependent on its context (Robshaw 1995); however, conventional stream ciphers can not be simply applied for encrypting the instructions in  $P$  since the instruction stream (execution sequence) of  $P$  varies according to conditional branches taken on its input. In our framework, through the process of translation  $P_0 \rightarrow P$ , we inject dummy instructions into  $P$  to force expedient state transitions in  $W$  so that  $P$  is always interpreted correctly regardless of its input.

Apart from obfuscation techniques, another possible way to hide secrets in software is program encryption (Albert and Morse 1984, Herzberg and Pinter 1987). Encrypting  $P_0$  by an encryption function  $E$  can make  $P_0$  difficult to understand. However, decryption  $E^{-1}$  must take place before executing an encrypted program  $E(P_0)$ , and this decryption must reveal  $P_0$  (or a part of  $P_0$ ) to the execution unit or interpreter, thus, hostile users have a chance to intercept and read the decrypted program  $P_0$ . On the other hand, in our framework, although  $W(P)$  may reveal an instructions stream of  $P_0$  for a particular input  $I$ , it will not reveal  $P_0$  itself. Anyway, obfuscation of code, obfuscation of interpretation, and encryption of code are not exclusive techniques, and could be used as complementary techniques to secure the software system.

The rest of this paper is organized as follows. In Section 2, a framework for obfuscated interpretation is proposed. Section 3 shows a case study of obfuscated interpretation. Section 4 discusses several attacks and defences. Finally, Section 5 concludes the paper with some suggestions for future work.

## 2 Framework for Obfuscated Interpretation

### 2.1 Overview

Before going into the mechanism of the FSM-based interpreter  $W$ , we describe the surroundings of  $W$  (Figure 2), then clarify the aim of our framework. The following are brief definitions of materials related to  $W$ .

$P_0$ : is a target program intended to be hidden from hostile users. For simplicity, we assume  $P_0$  is written in a low level programming language, such as bytecode or machine code, where each statement in  $P_0$  consists of a single opcode and (occasionally) some operands.

$W_0$ : is a common (conventional) interpreter for  $P_0$ , such as a Java Virtual Machine, a Common Language Runtime or an x86 processor.

$P_x$ : is a program containing obfuscated instructions whose semantics are determined during execution according to their context. This  $P_x$  is an equivalently translated version of  $P_0$ , i.e.  $P_x$  has the same functionality as  $P_0$ .

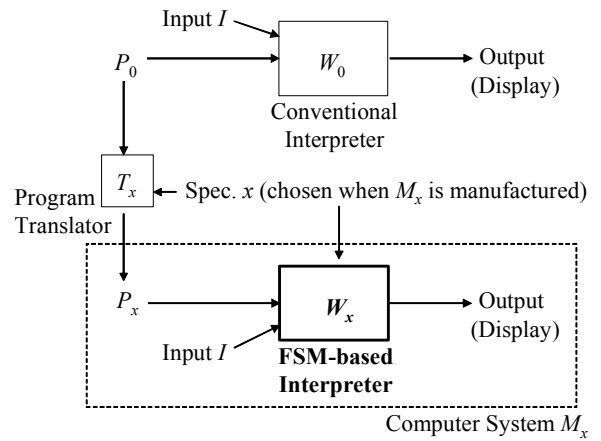


Figure 2. Framework for obfuscated interpretation

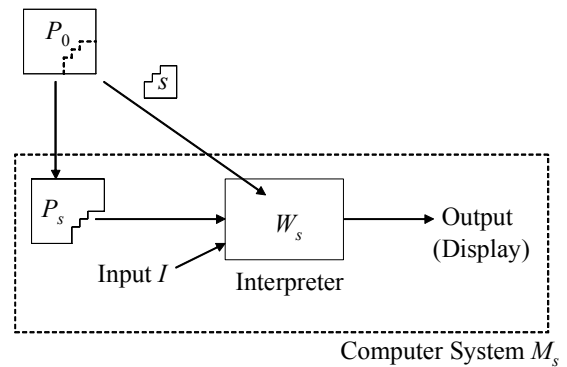


Figure 3. Alternative approach to hide program interpretation

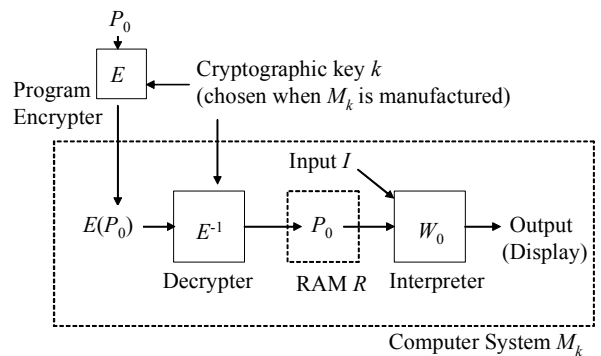


Figure 4. Basic approach for program encryption

$I$ : is an input of  $P_0$  and  $P_x$ . Note that  $P_0$  and  $P_x$  take the same input.

$x$ : is the specification of a FSM that defines a dynamic map between obfuscated instructions (inputs of the FSM) and their semantics (outputs of the FSM). This  $x$  is used both in a FSM-based interpreter  $W_x$  and a program translator  $T_x$ .

$W_x$ : is a FSM-based interpreter that can evaluate obfuscated instructions of  $P_x$  according to the current state of the FSM built inside. This  $W_x$  is an extension of  $W_0$  with a FSM unit of given specifications  $x$ .

$T_x$ : is a program translator that automatically translates  $P_0$  into  $P_x$  with respect to the specifications  $x$ .

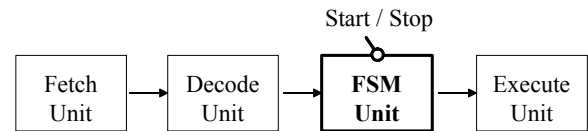
$M_x$ : is a computer system delivered to and/or owned by a program user.

In our framework, we assume  $W_x$  is hidden from the program user as much as possible, e.g. if  $M_x$  is an electronic device such as a mobile phone, then  $W_x$  should be built in a non-accessible part of  $M_x$  so as to prevent the user reading the implementation of  $W_x$ . However,  $P_x$  must be delivered to the user and put in an accessible area of  $M_x$  so as to enable its updating. There should be many functionally-different  $W_x$ , and ideally each machine  $M_x$  would be manufactured with a different  $W_x$  so that an adversary cannot easily guess one machine's interpreter after having "cracked" some other machine's interpreter.

Building an efficient  $T_x$  in a systematic manner is a fundamental part of this framework. Since  $P_x$  is quite different from ordinary programs, even though the program developer owns  $x$ , writing  $P_x$  from scratch is extremely difficult for the developer. In our framework, we provide a systematic method  $T_x$  to construct  $P_x$  from any given  $P_0$  and  $x$ .

In comparison to our framework, Figure 3 shows an alternative approach to hide the program interpretation from the user (T. Maude and D. Maude 1984, Zhang and Gupta 2003). In this approach, an essential piece of code (denoted  $s$ ) is cut off from  $P_0$ . This secret portion  $s$  is embedded in an interpreter  $W_s$  which is implemented in secure hardware, and attached to computer system  $M_s$ . The remaining part of the program (denoted  $P_s$ ) is delivered to the user in the usual way. This program is executed normally on the CPU in  $M_s$ , except for the secret portion which is executed by making calls to the interpreter  $W_s$ . For example, some of the arithmetic operations in  $P_s$  may be executed by  $W_s$ , possibly updating one or more state variables held in  $W_s$ . Since the return value from the calls to the interpreter  $W_s$  may be used to control branches and case statements in  $P_s$ , much of the control structure of  $P_0$  can be obscured. One difficulty with this approach is that it does not allow multiprogramming: while  $W_s$  is holding state for  $P_s$ , no other program can be run on  $W_s$ . Another problem is that any adversary who examines  $P_s$  will soon discover how to call  $W_s$ . The adversary can then write a program which makes similar calls to  $W_s$  in various orders. An analysis of the variability in the output of  $W_s$ , when it is exercised in this systematic way, is likely to reveal secrets of  $W_s$ . A final problem is that updates to  $P_s$  will, in general, require updates to its secret portion  $s$ . Thus we must have a secure channel for the transmission of  $s$  in encrypted form, and this channel is another avenue for attack. On the other hand, in our framework,  $W_x$  is built independent of  $P_0$ ; thus, many different programs run on a single interpreter  $W_x$ , and any of the programs can be easily updated without sending secret messages.

The most commonly-proposed method for hiding interpretation is program encryption. Figure 4 illustrates a typical scheme in which an encrypted program  $E(P_0)$  is delivered to the user, and a decrypter  $E^{-1}$  including a decryption key  $k$  is put in a non-accessible area of a computer system  $M_k$ . This  $E^{-1}$  decrypts  $E(P_0)$ , and puts the resultant  $P_0$  in a random-access memory  $R$ . Then, this  $P_0$  is passed to the interpreter  $W_0$  for execution. In this approach,



**Figure 5. Pipelined stages of FSM-based interpreter**

$E(P_0)$  itself is not understandable to the user. Also, many different programs can run on a single system  $M_k$ , and they are easily updatable. However, the problem of this approach is that it is not easy to completely hide the decrypted  $P_0$  from the user. One method for hiding the decrypted  $P_0$  is to decrypt only a small piece of  $E(P_0)$  at a time, and our approach takes this method to its logical extreme – we “decrypt” (translate) only one instruction at a time. Our approach minimises the size of RAM  $R$ , and building  $W_x$  in a non-accessible area of  $M_x$ 's hardware is easily realized by adding a small FSM unit to current hardware-based JVM implementations (such as picoJava, TinyJ, and Xpresso). A final point of distinction, as noted in Section 1 above, is that our interpreter translates the dynamic program stream, whereas decryption operates on the static representation of the program.

## 2.2 FSM-based interpreter

### 2.2.1 Design types

There are four types of design choices for the FSM-based interpreter, which are dependent upon the instruction set used for  $P_x$ . Let  $Ins_{P_0}$  and  $Ins_{P_x}$  be the instruction sets for  $P_0$  and  $P_x$ , and let  $L_{P_0}$  and  $L_{P_x}$  be the programming language for  $P_0$  and  $P_x$  respectively. We define four types of designs.

(Type 1)  $Ins_{P_x}$  is the same as  $Ins_{P_0}$  and all  $P_x$  have correct static semantics in  $L_{P_0}$  (e.g.  $P_x$  would pass Java's bytecode verifier if  $P_0$  were valid Java bytecode) although the dynamic semantics are determined during execution. Thus  $P_x$  is executable in the original interpreter  $W_0$  although its outputs would be incorrect.

(Type 2)  $L_{P_x}$  has the same syntax as  $L_{P_0}$ , but the stack signature of some opcodes in  $P_x$  may be incorrect. The number of different FSMs that could be used to interpret  $P_x$  is larger than in Type 1.

(Type 3)  $Ins_{P_x}$  includes  $Ins_{P_0}$  with some extra (“Type-3”) instructions. These may be used to control the FSM. The number of different FSMs is larger than in Type 2.

(Type 4)  $Ins_{P_x}$  differs completely from  $Ins_{P_0}$ , however there exists some (secret) many-to-one mapping which transforms  $Ins_{P_x}$  into a Type-3 instruction set. That is,  $P_x$  appears to be written in a totally different language than  $P_0$ . The number of different FSMs is larger than in Type 3.

In the rest of this paper, we focus on Type 2 designs.

### 2.2.2 Architecture

Figure 5 shows a suitable architecture for FSM-based interpreter, characterized by pipelined stages of interpretation. In Type 2 design, the FSM-based interpreter is augmented by an additional pipeline stage, called a *FSM unit*, which translates obfuscated instructions

and Type-2 syntax, producing output that is executable in a conventional *execute unit*. This architecture is easily applicable to many present virtual machines, such as JVMs (Java Virtual Machines) and CLR (Common Language Runtimes) of .NET.

The FSM unit has a switch to start/stop the obfuscated interpretation to enable us running both an ordinary program and an obfuscated program on the same interpreter. If the FSM unit is stopped, then the interpreter works as an ordinary one, and if it is started, then the interpreter works as a FSM-based interpreter. The start/stop signal could be invoked by a system call, or by a special Type 3 instructions.

### 2.2.3 FSM unit

The FSM unit (denoted as  $w_x$ ) is a DFA (Deterministic Finite Automaton) defined by 6-tuple  $(Q, \Sigma, \Psi, \Delta, A, q_0)$  where

$Q = \{q_0, q_1, \dots, q_{n-1}\}$  is the states in the FSM.

$\Sigma = \{c_0, c_1, \dots, c_{n-1}\}$  is the input alphabet.

$\Psi = \{\underline{c}_0, \underline{c}_1, \dots, \underline{c}_{n-1}\}$  is the output alphabet (interpretations for inputs).

$\delta_i : \Sigma \rightarrow Q$  is the next-state function for state  $q_i$ .

$\Delta = (\delta_0, \delta_1, \dots, \delta_{n-1})$  is the  $n$ -tuple of all next-state functions.

$\lambda_i : \Sigma \rightarrow \Psi$  is the output function for state  $q_i$ .

$A = (\lambda_0, \lambda_1, \dots, \lambda_{n-1})$  is the  $n$ -tuple of all output functions.

$q_0 \in Q$  is the starting state of the FSM.

In Type 2 design, the instruction set for  $P_x$  is the same as that for  $P_0$ , so  $Ins_{P_x} = Ins_{P_0}$ . We assume  $Ins_{P_x} = \Sigma \cup O$  where elements  $c_i \in \Sigma$  are obfuscated instructions, and  $o_i \in O$  are non-obfuscated instructions. This means,  $P_x$  contains both  $c_i$  and  $o_i$ , and, if the FSM unit recognizes  $c_i \in \Sigma$  as input then its semantics is determined by the FSM and it is passed to the execute unit, otherwise an input  $o_i \in O$  is directly passed to the execute unit. For simplicity, this paper focuses on opcodes to be translated in the FSM.

In our Type-2 design, each underlined symbol  $\underline{c}_i$  in  $\Psi$  denotes the normal (untranslated) semantics for the correspondingly-indexed opcode  $c_i$  in  $\Sigma$ .

The input (and output) alphabet is partitioned into two classes by an integer  $b$ , such that symbols  $c_0, c_1, \dots, c_{b-1}$  are in the first class  $C_1$  (of branching opcodes including non-conditional jump) and the remaining symbols  $c_b, c_{b+1}, \dots, c_{n-1}$  are in the second class  $C_2$  (of non-branching opcodes).

Our FSM design has the following constraints.

1. Each  $\delta_i : \Sigma \rightarrow Q$  is a bijection; we will use its inverse  $\delta_i^{-1} : Q \rightarrow \Sigma$ .
2. Each  $\lambda_i : \Sigma \rightarrow \Psi$  is a bijection, defining  $\lambda_i^{-1} : \Psi \rightarrow \Sigma$ .
3. For all  $i$  and  $j$ ,  $\lambda_i(c_j)$  has the same operand signature as  $c_j$  (but not necessarily the same stack signature). For example, "push x" has the same operand signature as "pop x" but it differs from "add x, y".
4. For all pairs of states  $q_i, q_k$  there exists a "dummy instruction sequence"  $\underline{d}_i$  with the following three properties. First,  $\underline{d}_i$  is a short sequence of (translated)

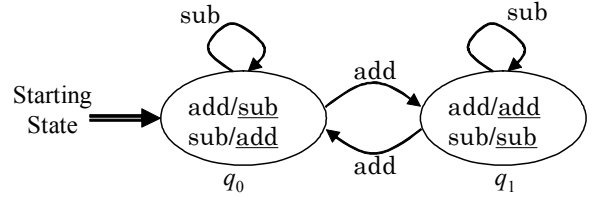


Figure 6. Example of FSM  $w_x$

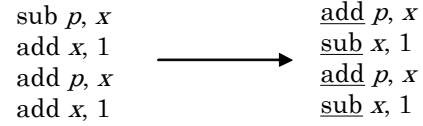


Figure 7. Example of instruction stream interpretation

```

let x = n
let p = 1
loop: if x == 0 exit
      add p, x
      sub x, 1
      goto loop:

```

Figure 8. Example of  $P_0$

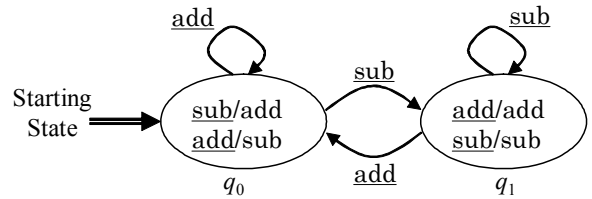


Figure 9. Example of  $w_x^{-1}$

instructions containing exactly one obfuscated instruction. Second, an FSM initially in state  $q_i$  will be in state  $q_k$  after it produces  $\underline{d}_i$  as output. Third,  $\underline{d}_i$  has no effective functionality. Thus  $\underline{d}_i$  is an efficiently executed no-op that forces the FSM to make any desired transition. Note that for any pair of states  $q_i, q_k$  there exists  $c_z$  such that  $\delta_i(c_z) = q_k$ , because the next-state function  $\delta_i$  is a bijection. The obfuscated instruction in  $\underline{d}_i$  is  $\lambda_i(c_z)$ .

5. For all states  $q_k$  and branching instructions  $c_j \in C_1$ , there exists a state  $q_i$  with the property  $\delta_i(c_j) = q_k$ . That is, if we have a branching instruction  $c_j$  and a desired state  $q_k$  to be reached, we can find some initial state  $q_i$  that reaches  $q_k$  via the input  $c_j$ . (When we translate a branch instruction  $c_j$ , we apply the previous constraint to force the FSM into state  $q_i$  if the instruction at the target of the branch must be interpreted in state  $q_k$ .)

Figure 6 shows a simple example of  $w_x$  where

$Q = \{q_0, q_1\}$   
 $\Sigma = \{\text{add, sub}\}$   
 $\Psi = \{\underline{\text{add}}, \underline{\text{sub}}\}$   
 $\Delta : \delta_0(\text{add}) = q_1, \delta_0(\text{sub}) = q_0, \delta_1(\text{add}) = q_0, \delta_1(\text{sub}) = q_1$   
 $A : \lambda_0(\text{add}) = \underline{\text{sub}}, \lambda_0(\text{sub}) = \underline{\text{add}}, \lambda_1(\text{add}) = \underline{\text{add}}, \lambda_1(\text{sub}) = \underline{\text{sub}}$

This  $w_x$  takes an opcode  $c_i \in \{\text{add}, \text{sub}\}$  as an input, translates it into its semantics  $\underline{c}_i \in \{\underline{\text{add}}, \underline{\text{sub}}\}$ , and outputs  $\underline{c}_i$ . Figure 7 shows an example of interpretation for an instruction stream done by this  $w_x$ . Obviously, even this simple FSM has the ability to conduct the obfuscated interpretation. As shown in Figure 7, the opcode “add” is interpreted as either add or sub according to its context.

We can build a much more complex  $w_x$  on the larger  $\Sigma$  of actual programming languages. Using two different FSM units, one for opcodes and the other for operands, we could build even more complex Type 2 designs.

## 2.2.4 Program translator

In order to utilize the FSM-based interpreter  $W_x$ , a program translator  $T_x: P_0 \rightarrow P_x$  is indispensable. However, building  $T_x$  is much more than building an inverse interpreter of  $w_x$ . Let us assume we have  $w_x$  of Figure 6, and  $P_0$  of Figure 8 that computes a summation  $p := 1+2+3+\dots+n$ . The loop in  $P_0$  must be taken into account. We need a consistency of interpretation: the instructions in each execution of the loop in  $P_x$  must always be translated into the same instruction stream (in this case, “add p, x” and “sub x, 1”). In other words,  $w_x$  must always be in the same state every time the execution reaches the control-flow junction at the top of the loop body. Taking advantage of constraints 4 and 5 above, we inject a sequence of dummy instructions into the tail of the loop, so that the FSM will reach the desired state at the top of the loop without changing program semantics.

Anyway, we first build an inverse interpreter of  $w_x$  (denoted as  $w_x^{-1}$ ), then we use this inverse interpreter to translate  $P_0$  into  $P_x$ . Our  $w_x^{-1}$  is the DFA defined by 6-tuples  $(Q', \Sigma', \Psi', \Delta', A^{-1}, q_0)$  where

- $Q' = Q = \{q_0, q_1, \dots, q_{n-1}\}$  is the states in the FSM.
- $\Sigma' = \Psi = \{\underline{c}_0, \underline{c}_1, \dots, \underline{c}_{n-1}\}$  is the input alphabet.
- $\Psi' = \Sigma = \{c_0, c_1, \dots, c_{n-1}\}$  is the output alphabet.
- $\delta_i' : \Sigma' \rightarrow Q'$  is the next-state function for state  $q_i$ , where  $\delta_i'(\underline{c}_i)$  has the value  $\delta_i(\lambda_i^{-1}(\underline{c}_i))$  for all  $i, j$ .
- $\Delta' = (\delta_0', \delta_1', \dots, \delta_{n-1}')$  is the  $n$ -tuple of all next-state functions.
- $\lambda_i' : \Sigma' \rightarrow \Psi'$  is the output function for state  $q_i$ , where each  $\lambda_i' : \Sigma' \rightarrow \Psi'$  has the value  $\lambda_i^{-1}(\underline{c}_i)$  for all  $i, j$ .
- $A^{-1} = (\lambda_0', \lambda_1', \dots, \lambda_{n-1}')$  is the  $n$ -tuple of all output functions.
- $q_0 \in Q'$  is the starting state of the FSM.

Figure 9 shows an example of  $w_x^{-1}$  corresponding to  $w_x$  of Figure 6. As shown in Figure 9,  $w_x^{-1}$  has the same number of states and transitions as  $w_x$ .

Next, we give a procedure for the translation  $T_x: P_0 \rightarrow P_x$ . Figure 10 shows this procedure where:

- $PC$  is a program counter (we assume  $PC$  is a line number of  $P_0$ ).
- $code_{p_0}(PC)$  is an instruction in  $P_0$  at  $PC$ .
- $code_{p_x}(PC)$  is an instruction in  $P_x$  at  $PC$ .
- $q_s \in Q$  is a state of  $w_x^{-1}$ .
- $state(PC)$  is a state in which  $code_{p_0}(PC)$  was interpreted.

```

Let  $state(k) := \text{NULL}$  for all  $k$  of  $P_0$ 
Let  $q_s := q_0$ 
Set  $PC$  to the entry point of  $P_0$ 
loop:
If  $PC = \text{exit}$  of  $P_0$  then goto resume
If  $state(PC) \neq \text{NULL} \ \&\& \ state(PC) \neq q_s$  then {
  Call choose&insert_dummy
  Goto resume
}
Let  $state(PC) := q_s$ 
If  $code_{p_0}(PC) \in \Sigma'$  then { /* obfuscated instruction */
  Interpret  $code_{p_0}(PC)$  via  $w_x^{-1}$ , i.e.
  Let  $q_s := \delta_s'(code_{p_0}(PC))$ 
  Let  $code_{p_x}(PC) := \lambda_s^{-1}(code_{p_0}(PC))$ 
} else { /* non-obfuscated instruction */
  Let  $code_{p_x}(PC) := code_{p_0}(PC)$ 
}
If  $code_{p_0}(PC) = \text{branching instruction}$  then {
  If  $code_{p_0}(PC) \neq \text{non-conditional jump}$  then {
    Do  $push(PC_{\text{false}})$  where  $PC_{\text{false}}$  is a line number of
    next instruction in  $false$  branch
    Let  $state(PC_{\text{false}}) = q_s$ 
  }
  Let  $PC :=$  a line number of next instruction in  $true$ 
  branch
} else {
   $PC := PC + 1$ 
}
Goto loop

resume:
If  $Stack$  is empty then end
 $PC := pop()$ 
 $q_s := state(PC)$ 
Goto loop

choose&insert_dummy:
Let  $PC_{\text{prev}} :=$  previous value of  $PC$ 
If  $code_{p_0}(PC_{\text{prev}}) = \text{non-branching instruction}$  then {
  Choose  $\underline{c}_i \in \Sigma'$  that satisfies  $\delta_s'(\underline{c}_i) = state(PC)$ 
  Let  $\underline{d}_i :=$  a sequence of dummy instructions for  $\underline{c}_i$ 
  Let  $d_i := \lambda_s^{-1}(\underline{d}_i)$ 
  Insert  $d_i$  into  $P_x$  right after the line number =  $PC_{\text{prev}}$ 
} else {
  Choose  $k$  that satisfies  $\delta_k'(code_{p_0}(PC_{\text{prev}})) = state(PC)$ 
  Choose  $\underline{c}_i \in \Sigma'$  that satisfies  $\delta_{state(PC_{\text{prev}})}'(\underline{c}_i) = q_k$ 
  Let  $\underline{d}_i :=$  a sequence of dummy instructions for  $\underline{c}_i$ 
  Let  $d_i := \lambda_s^{-1}(\underline{d}_i)$ 
  Insert  $d_i$  into  $P_x$  at the line number =  $PC_{\text{prev}}$ 
   $state(PC_{\text{prev}}) = q_k$ 
}
return

```

**Figure 10. Procedure for  $T_x: P_0 \rightarrow P_x$**

We also assume this procedure  $T_x$  uses a stack (denoted as  $Stack$ ), and its operation  $push$  and  $pop$ , to accumulate values of  $PC$ .

Figure 11 shows an example of  $P_x$  translated from  $P_0$  of Figure 8. In this example, a dummy instruction “add p, 0” is inserted into  $P_x$  to force the state transition  $q_1 \rightarrow q_0$  so that  $w$  comes to  $q_0$  every time the execution reaches the entry point of loop.

```

let x = n
let p = 1
loop: if x == 0 exit
sub p, x
add x, 1
add p, 0 ; dummy instruction
goto loop:

```

Figure 11. Example of translated  $P_x$

```

static int sumodd(int N){
    int i, p;
    p = 0;
    for(i = 1; i <= N; i++){
        if(i % 2 == 1) p = p + i;
    }
    return p;
}

```

Figure 12. Example of  $P_0$  in Java

### 3 Case Study

#### 3.1 Program translation

In this section we explain a more complex example, in which we execute the procedure  $T_x: P_0 \rightarrow P_x$  of Figure 10 using the inverse interpreter  $w_x^{-1}$  given in Table 1. This  $w_x^{-1}$  has eight states  $Q' = \{q_0, q_1, \dots, q_7\}$  with  $q_0$  a starting state, and has eight instructions  $\Sigma' = \{\text{goto}, \text{if icmpne}, \text{iload 1}, \text{iconst 1}, \text{iconst 2}, \text{iadd}, \text{iload 2}, \text{irem}\}$ . Two instructions (goto and if icmpne) are branching instructions having one operand, and rest of six are non-branching instructions having no operand.

Table 2 shows sequences of dummy instructions  $\underline{d}_i$  for each  $c_i \in \Sigma'$ . The obfuscated (translated) dummy sequence  $d_i = \lambda_j^{-1}(\underline{d}_i)$  does not change the behaviour of  $P_x$ , yet it causes one state-transition in  $w_x$ .

The target  $P_0$ , which is to be translated, is shown in Figure 13. This  $P_0$  is a Java bytecode program described in *jasmin* format (Meyer and Downing 1997). Source code of  $P_0$  is shown in Figure 12. This  $P_0$  computes a summation of odd numbers  $p := 1+3+5+\dots+n$ . Figure 14 shows  $P_x$  corresponding to this  $P_0$ . In Figure 13, numbers described in leftmost column indicates line numbers, and their corresponding lines are described in Figure 14 as well. Second column in Figure 13 describes the state of  $w^{-1}$  in which each instruction is interpreted.

The translation starts at state  $q_s := q_0$ . In line 1 and 2 of  $P_0$ , instructions "iconst 0" and "istore 2" are not translated since they are not in  $\Sigma'$ . In line 3, "iconst 1" is translated into "iload 1", and  $q_2$  becomes the current state of  $w_x^{-1}$ . Each time a new line is reached, let  $state(line\ number)$  be the current state. In line 4, "istore 1"  $\notin \Sigma'$  is not translated. Then, in line 5, "goto Label3" is translated into "if icmpne" and  $q_7$  becomes the current state, and with this "goto Label3" we jump to the Label3 at line 18.

After the interpretation of line 19 and 20, a branch instruction "if icmple Label1" in line 21 is reached; then, the line number of false branch "22" is pushed onto the stack, let  $state(22) := q_s$ , and we jump to the true branch

"Label1" in line 6. Similarly, after the interpretation of line 7-10, when we reach "if icmpne Label2" in line 11, this is translated into "goto Label2", then the line number of false branch "12" is pushed onto the stack, let  $state(12) := q_0$ , and we jump to the true branch "Label2" in line 16.

Shortly, line 19 is reached for the second time. Recalling when we visited this line for the first time, the state  $q_s$  was  $q_7$ , thus,  $state(19) = q_7$ . However, in this second visit  $q_s = q_0$ , so now  $state(19) \neq q_s$ . Hence, we need to add a sequence of dummy instructions  $d_i$  to  $P_x$  to force the transition  $q_0 \rightarrow q_7$  in  $w_x$ . In order to choose  $d_i$ , firstly we choose  $\underline{c}_i \in \Sigma'$  that satisfies  $\delta_{q_0}(\underline{c}_i) = q_7$ , i.e. choose an instruction that force the transition  $q_0 \rightarrow q_7$  in  $w_x^{-1}$ . Obviously from Table 1,  $\underline{c}_i = \text{"if icmpne"}$ . Next, we choose  $\underline{d}_i$  for  $\underline{c}_i$ . Looking at #1 of Table 2, we choose  $\{\text{bipush 1}, \text{if icmpne}\}$  as  $\underline{d}_i$ . Afterward, we obtain its obfuscated sequence  $d_i := \lambda_{q_0}^{-1}(\underline{d}_i) = \{\text{bipush 1}, \text{goto}\}$ . Finally, we inject this  $d_i$  between line 17 and 18 of  $P_x$  (Figure 14). Note that an arbitrary operand (a label for an instruction) can be applied to the opcode "goto" in  $d_i$ . In this example, "Label4" is inserted at line 10 and used as the operand.

After injecting  $d_i$ , the interpretation is interrupted since the instruction in line 19 is already translated. Then, a value 12 is popped from the stack, let  $q_s := state(12) = q_0$ , and the translation resumes from line 12.

Finally, end of the program is reached, and the stack is now empty. Thus, the translation is finished.

#### 3.2 Obscurity of translated program

The program  $P_x$  obtained by above translation has some fundamental characteristics to make itself obscure. Below we describe the characteristics of  $P_x$  in Figure 14 compared with  $P_0$  in Figure 13.

1. As described in 2.2.1,  $P_x$  has the same syntax as Java bytecode, but the stack signature of some opcodes in  $P_x$  may be incorrect. For example, in line 5, "if icmpne" instruction requires one integer value to be popped from the operand stack of the JVM, however, the stack is empty at line 5.
2. Instructions in  $P_x$  do not have static binding to their semantics. For example, "iload 1" in line 3 is interpreted as "iconst 1" via  $w_x$  (see the same line in Figure 13), but in line 9, it is interpreted as "irem". Note that a part of dummy instructions also have non-static semantics.
3. The control flow of  $P_x$  is not preserved, i.e. different from that of  $P_0$ . For example, the non-conditional jump "goto" in line 11 is interpreted as a conditional jump "if icmpne". In addition, "goto" instruction between line 17 and 18 seems to jump to Label4, however, actually it is translated into a dummy "if icmpne".

#### 3.3 Program execution

There is one significant restriction on executing  $P_x$  in the computer system  $M_x$ . That is, a bytecode verifier cannot be

```

1  q0  iconst 0
2  q0  istore 2
3  q0  iconst 1
4  q2  istore 1
5  q2  goto Label3
6  q2  Label1:
7  q5  iload 1
8  q4  iconst 2
9  q3  irem
10 q0  iconst 1
11 q2  if_icmpne Label2
12 q0  iload 2
13 q3  iload 1
14 q2  iadd
15 q3  istore 2
    q3
16 q3  Label2:
17 q0  iinc 1 1
    q0
18 q0  Label3:
19 q7  iload 1
20 q5  iload 0
21 q5  if_icmple Label1
22 q5  iload 2
23 q5  ireturn

```

**Figure 13.**  $P_0$  in jasmin format

used since  $P_x$  is not a valid Java bytecode program. Obviously,  $P_x$  in Figure 14 is not “stack balanced”. This suggests only trustworthy programs should be run on  $M_x$ . However, since the FSM specification  $x$  must be kept secret from users, only licensed (i.e. trusted) developers are allowed to develop  $P_x$ , thus, this restriction may not be so serious. Anyway, if we want to use the Java bytecode verifier, we can use Type 1 interpreter of Section 2.2.1.

#### 4 Security Analysis

In this section, we analyze the security of our scheme against adversaries of varying resources, knowledge, and persistence.

Generally speaking, our security objective is to prevent an adversary from understanding the protected software. The understanding of an adversary is not directly measurable, however, so we define our security metric by a series of restrictions on an adversary’s future actions.

1. [Local tamper-proofing] The adversary should not understand the protected software well enough to make small alterations in program representation and behavior. An example of a small alteration is the replacement of an IFNE opcode with a GOTO opcode, in order to defeat a license check (LaDue 1997).
2. [Global tamper-proofing] The adversary should not understand the protected software well enough to make large-scale alterations in representation and/or small alterations in behavior. An example of a large-scale alteration in representation is a de-compilation and re-compilation. Such an attack will obscure many static code watermarks (Collberg and Thomborson 2002), and it will defeat a copyright-violation test that is based on a code comparison.
3. [Reverse engineering; algorithmic understanding] The adversary should not understand the protected

```

1  iconst 0
2  istore 2
3  iload 1
4  istore 1
5  if_icmpne Label3
6  Label1:
7  irem
8  iload 2
9  iload 1
10 iload 1
    Label4:
11 goto Label2
12 iadd
13 iconst 1
14 iadd
15 istore 2
    bipush 1 ; dummy
    bipush 1 ; dummy
    iload 1 ; dummy q3→q0
    pop ; dummy
16 Label2:
17 iinc 1 1
    bipush 1 ; dummy
    goto Label4 ; dummy q0→q7
18 Label3:
19 iconst 1
20 iload 0
21 if_icmple Label1
22 iadd
23 ireturn

```

**Figure 14.**  $P_x$  in jasmin format

software well enough to make a large-scale alteration in its behavior, for example by identifying, copying, and re-using a substantial portion of its code (or its embedded “secrets” such as a decryption key) in another software product.

We have listed these restrictions in order of increasing understanding. Only an adversary with “level-3 understanding”, in our metric, is able to reverse-engineer a program. Such an adversary would also possess level-2 and level-1 understanding. An adversary who has level-2 understanding can de-compile (or at least dis-assemble) the code, and then make wholesale changes in program representation and some changes in behavior. An adversary with level-1 understanding may discover, through a trial-and-error process, a conditional branch whose annulment will defeat a simple license-checking mechanism.

Below we will show that our protection scheme (in conjunction with other obfuscations) will prevent adversaries with considerable knowledge, resources and motivation from gaining level-3 understanding. Weaker adversaries will not gain level-2 or even level-1 understanding, unless they are very persistent.

We characterize an adversary by the software tools they have available, see list “A” below. These tools define the types of observation steps (see list “B”) and control steps (“C”) our adversaries can make. Our security analyses are bounds on the number of steps taken by a given adversary to reach a given level of understanding.

##### A. Adversaries.

0. Our level-0 adversary is a naïve end-user with a computer system  $M_x$  (containing interpreter  $W_x$  as

shown in Figure 2) and a copy of the translated (protected) program  $P_x$ . Note that these resources are required to execute the protected program.

1. Our level-1 adversary has some skills in computer science and cryptography, coupled with an algorithmic understanding of the principles of FSM-based interpretation, as described in this article.
  2. Our level-2 adversary is expert and extremely well-resourced, possessing a debugger with “breakpoint” functionality, attached to an obfuscated software implementation of  $W_x$ . Alternatively, the adversary has a logic state analyzer, attached to the inputs and outputs of a hardware implementation of  $W_x$ .
  3. Our level-3 adversary has specialised tools allowing them to collect output traces from  $W_x$ , and to inject arbitrary inputs for translation by  $W_x$ .
  4. Our level-4 adversary has a generic interpreter  $W(x)$  which emulates  $W_x$  for all  $x$ , and also a generic translator  $T(x)$  which implements  $T_x$  for all  $x$ .
- B. Observation.
0. In a level-0 observation, the adversary observes the audio-visual outputs of the computer system  $M_x$ , as it executes a program.
    1. The adversary determines, by inspection of audio-visual outputs, whether or not  $M_x$  is running a program that has the same behavior as the protected program.
    2. The adversary records a snapshot (*i.e.* a small number of opcodes and operands, before and after FSM interpretation) of the input and output of  $W_x$ .
    3. The adversary records a complete trace of the output of  $W_x$ , during a run of the protected program on computer system  $M_x$ .
    4. The adversary uses the level-4 generic translator  $T(x)$  to obtain a translation of a program  $P$  for FSM  $x$ . Alternatively, the adversary uses  $W(x)$  to obtain a complete output trace.
- C. Control.
0. The adversary operates the keyboard and mouse inputs of the computer system  $M_x$ , as it executes the protected program.
    1. The adversary can modify the statements in program  $P_x$  in any desired way, before running it on computer system  $M_x$ .
    2. The adversary injects a small number of (arbitrary) inputs into  $W_x$ , after the unit has interpreted some (arbitrary) number of opcodes and operands. These injections are at low speed, and for this reason they will generally not produce the same audio-visual output from system  $M_x$  as if these inputs were normally presented to  $W_x$ .
    3. The adversary injects arbitrary inputs into  $W_x$ , at full bandwidth.
    4. The adversary injects arbitrary inputs, including the setting of parameter  $x$ , into  $T(x)$  and  $W(x)$ .

Under our definitions above, level-0 adversaries have very few avenues of attack. They might attempt a “black-box re-engineering” – inferring program code from program behavior. Such an attack is infeasible unless program behavior is trivial, and in any event it would not breach any of our security objectives.

The only other avenue of attack of a level-0 adversary is an inspection and cryptographic analysis of the translated program  $P_x$ . An early step in such an analysis would be a working knowledge of the principles of FSM interpretation, which would be much more effectively gained by reading this article (a level-1 attack) than by a naïve level-0 attack.

We turn to the level-1 attacks. A cryptographically-skilled adversary with knowledge of programming language semantics and our FSM algorithm would probably start by building a table of “dummy instruction sequences”  $\underline{d}_i$  similar to Table 2. Note that the obfuscation on these sequences is weak. Each dummy sequence consists of a short (possibly empty) prefix of non-obfuscated instructions, a single obfuscated instruction, and a short (possibly empty) suffix of non-obfuscated instructions. Algorithm  $T_x$  will place a dummy instruction sequence at the end of branch to a predecessor instruction, except in the (relatively rare) cases where the FSM is in the same state in both paths to the target instruction. So the suffixes will be recognizable as the commonly-repeated patterns before a backwards-branch or jump. Note that all control-flow opcodes are recognizable (either as class- $C_1$  opcodes, or as unobfuscated opcodes) because of our constraint on operand signatures. The adversary might have to examine  $O(n^2)$  loops to be reasonably certain of having discovered all suffixes, so hypothesizing  $\underline{d}_i$  might take days but not months if  $n = 100$ . The prefixes can be recognized as the commonly-repeated short sequences that occur immediately before a single (variable) instruction that precedes a suffix. The attacker can prune the list of possible dummy sequences by discarding any prefix-suffix pair that is not a no-op for at least one choice of (variable) instruction semantics.

Our level-1 attacker would continue their attack by building up a (hypothesized) list  $d_{jk}$  of obfuscated dummy sequences by substituting all (hypothesized)  $c_k$  for the  $\underline{c}_i$  in each (hypothesized) sequence  $\underline{d}_i$ . Using their level-1 control, they could insert an arbitrary instruction at the beginning of a (hypothesized) loop body; this will soon reveal the location of a sensitive loop, whose semantics visibly affects program operation (a level-1 observation). The attacker would then insert a short no-op sequence to confirm that program correctness is not hypersensitive to loop timing. Then the attacker would choose one pair  $d_{ij}$ ,  $d_{kz}$  of the (hypothesized) obfuscated dummy sequences for insertion at this point in the program. A small fraction of these pairs (about 1/10000 if there are 100 obfuscated opcodes) will not affect program correctness. One such discovery constitutes a major “crack” because the attacker is almost certain that the FSM was in the same state at the beginning and the end of this sequence. After 10000 such discoveries, the attacker would have cracked a 100-state FSM  $W_x$ . We have not done a complete cryptographic analysis, however our preliminary analysis indicates that  $O(n^4)$  observations and controls would suffice for an attack of the type described above, on an  $n$ -state Type 2 FSM by an extremely persistent level-1 attacker with cryptographic skill. This might take months or years, because each step requires our adversary to observe a run of a modified  $P_x$  on their machine  $M_x$ . We could increase the difficulty of such attacks by increasing the search space: using multiple

“dummy sequences” for each instruction; or relaxing our constraint on operand signatures so that branching opcodes are not immediately recognisable; or using a Type-3 FSM to make it harder for the attacker to recognize no-op suffixes and prefixes; or using a Type-4 FSM to increase  $n$ . These are topics for our future research.

The “crack” described above for a level-1 attacker gives them a level-2 understanding of a single machine  $M_x$ , for they can now predict how a single FSM  $W_x$  will translate arbitrary inputs – including the obfuscated program  $P_x$ ! (Note: the attacker must also do some cut-and-paste work, and some exercising of program paths, perhaps by program modification, to transform their traces of  $P_x$  into a program listing. Alternatively, they might choose to write source code for a specialised de-obfuscator  $T_x$ : this may take months, but they have probably already spent months if not years to reach this level of understanding: they are now essentially a level-3 adversary.) The level-1 adversary in possession of this “crack” can also discover the FSM state at any point in the code where their code insertions can visibly affect program correctness. With this knowledge they can inject short code sequences, followed by an appropriate “dummy sequence” to preserve the correctness of translation of the subsequent code.

We now briefly consider level-2 and level-3 adversaries.

A level-2 adversary can correlate the outputs with the inputs of the FSM, where these inputs are the ones associated with any desired “breakpoint” in a (possibly modified)  $P_x$ . This ability will greatly speed the brute-force attack described above for our level-1 adversary, and it will allow new attack strategies such as directly observing the translation  $\lambda_i(c_z)$  of an instruction  $c_z$  that occurs in (hypothesised) dummy sequences in  $P_x$ . Our preliminary analysis indicates that  $O(n^3)$  observations and controls, each taking a few seconds or milliseconds (in an automated attack), will suffice for a level-2 attacker to achieve level-2 understanding.

A level-3 adversary can collect an execution trace of  $P_0$ , and they can correlate all branch-points in this trace with the corresponding branch-points in  $P_x$ . If  $P_x$  is short, they can produce an accurate cleartext bytecode listing by hand. If  $P_x$  is long, they should try to obtain a copy of a “general-purpose” deobfuscating tool that some other level-3 adversary may have produced when cracking some other  $M_x$ . If no such tool exists, our level-3 attacker may write and publish such a tool, so that subsequent level-3 attackers merely have to obtain the tool to get a program listing for any  $P_x$ . However we note that, if the value of  $x$  is embedded in secure hardware, and if the party in possession of  $T_x$  preserves the secrecy of  $x$ , level-3 adversaries will be rare – they must either have the ability to “crack” secure hardware or they must develop more powerful cryptanalytic attacks than we have outlined above for our hypothetical level-2 adversary.

We close our security analysis with a warning. Our translation system is essentially cryptographic in nature, so it should only be used to obfuscate long programs that have been “randomized” (*i.e.* obfuscated) before they are translated by our  $T_x$ . Otherwise the attacker will be able to

make a likely guess to the cleartext, which may greatly speed their attack.

## 5 Conclusion

In this paper we proposed a framework for obfuscating the program interpretation. We defined a FSM-based interpreter  $w_x$  that gives context-dependent semantics to program instructions. We also defined a program translator  $T_x$  to systematically construct a program  $P_x$ , which is executable with  $w_x$ , from a given program  $P_0$  written in a conventional programming language.

Our case study of a “Type 2” translation of  $P_0$  into  $P_x$  showed that instructions in  $P_x$  have non-static semantics, *i.e.* functionality is hidden from program users, yet  $P_x$  is still functionally equivalent to  $P_0$ .

Our preliminary security analysis showed that our design is reasonably secure against adversaries of varying resources, knowledge, and persistence. Our analysis highlighted some areas where our design could be improved, and we conclude that our design should only be used to obfuscate long programs that have been “randomized” (*i.e.* obfuscated) before they are translated.

In the future, we will develop detailed designs for Type 1, 3 and 4 interpreters, and we intend to clarify their advantages and shortcomings.

## 6 References

- Albert, D.J. and Morse, S.P. (1984): Combatting software piracy by encryption and key management. *Computer*, **17**(4):68-73, IEEE Computer Society.
- Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai A., Vadhan, S. and Yang, K. (2001): On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, **2139**:1-18, Springer-Verlag.
- Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanovic, D. and Zovi, D.D. (2003): Randomized instruction set emulation to disrupt binary code injection attacks. *Proc. 10th ACM Conference on Computer and Communications Security (CCS2003)*, 281-289, Washington DC, USA.
- Chow, S., Eisen, P., Johnson, H. and Oorschot, P.C. van (2002): A white-box DES implementation for DRM applications. *Proc. 2nd ACM Workshop on Digital Rights Management (DRM2002)*, Washington DC, USA.
- Cohen, F.B. (1993): Operating system protection through program evolution. *Computers and Security*, **12**(6):565-584, Elsevier Science.
- Collberg, C. and Thomborson, C. (2002): Watermarking, tamper-proofing, and obfuscation - Tools for software protection. *IEEE Transactions on Software Engineering*, **28**(8):735-746, IEEE Computer Society.
- Herzberg, A. and Pinter, S.S. (1987): Public protection of software. *ACM Transactions on Computer Systems*, **5** (4):371-393.

Kanzaki, Y., Monden, A., Nakamura, M. and Matsumoto, K. (2003): Exploiting self-modification mechanism for program protection. *Proc. 27th IEEE Computer Software Applications Conference (compsac2003)*, 170-179, Dallas, USA, Nov. 2003.

Kc, G.S., Keromytis, A.D. and Prevelakis, V. (2003): Countering code-injection attacks with instruction-set randomization. *Proc. 10th ACM Conference on Computer and Communications Security (CCS2003)*, 272-280, Washington DC, USA.

LaDue, M. (1997): The Maginot license: Failed approaches to licensing Java software over the Internet. <http://www.geocities.com/securejavaapplets/maginot.html>

Maude, T. and Maude, D. (1984): Hardware protection against software piracy. *Communications of the ACM*, 27(9):950-959.

Meyer, J. and Downing T. (1997): *Java Virtual Machine*. O'Reilly & Associates, Inc.

Robshaw, M.J.B. (1995): Stream ciphers. *RSA Laboratories Technical Report TR-701*. <http://islab.oregonstate.edu/koc/ece575/rsalabs/tr-701.pdf>

Sander, T. and Tschudin, C.F. (1998): Protecting mobile agents Against malicious hosts. *Lecture Notes in Computer Science*, 1419:44-60, Springer-Verlag.

The U.K. Parliament (2002): The mobile telephones (re-programming) bill. *House of Commons Library Research Paper 02-47*.

<http://www.parliament.uk/commons/lib/research/rp2002/rp02-047.pdf>

Zhang, X. and Gupta, R. (2003): Hiding program slices for software security, *Proc. International Symposium on Code Generation and Optimization (CGO2003)*, 325-336, San Francisco, USA.

**Table 2. List of sequence of dummy instructions**

Sequence of dummy instructions	#
<b>goto Label x</b> any instructions Label x	0
<b>bipush 1</b> <b>if icmpne</b>	1
<b>iload 1</b> <b>pop</b>	2
<b>iconst 1</b> <b>pop</b>	3
<b>iconst 2</b> <b>pop</b>	4
<b>bipush 0</b> <b>iadd</b>	5
<b>iload 2</b> <b>pop</b>	6
<b>bipush 1</b> <b>bipush 1</b> <b>irem</b> <b>pop</b>	7

**Table 1. Example of FSM  $w_x^{-1}$**

State	Input / Output	transition	State	Input / Output	transition
$q_0$	<b>goto</b> / <b>if icmpne</b>	$q_4$	$q_4$	<b>goto</b> / <b>if icmpne</b>	$q_5$
	<b>if icmpne</b> / <b>goto</b>	$q_7$		<b>if icmpne</b> / <b>goto</b>	$q_4$
	<b>iload 1</b> / <b>iconst 2</b>	$q_5$		<b>iload 1</b> / <b>iconst 2</b>	$q_2$
	<b>iconst 1</b> / <b>iload 1</b>	$q_2$		<b>iconst 1</b> / <b>irem</b>	$q_1$
	<b>iconst 2</b> / <b>iconst 1</b>	$q_1$		<b>iconst 2</b> / <b>iload 2</b>	$q_3$
	<b>iadd</b> / <b>irem</b>	$q_6$		<b>iadd</b> / <b>iload 1</b>	$q_0$
	<b>iload 2</b> / <b>iadd</b>	$q_3$		<b>iload 2</b> / <b>iconst 1</b>	$q_7$
<b>irem</b> / <b>iload 2</b>	$q_0$	<b>irem</b> / <b>iadd</b>	$q_6$		
$q_1$	<b>goto</b> / <b>goto</b>	$q_1$	$q_5$	<b>goto</b> / <b>goto</b>	$q_0$
	<b>if icmpne</b> / <b>if icmpne</b>	$q_3$		<b>if icmpne</b> / <b>if icmpne</b>	$q_1$
	<b>iload 1</b> / <b>iconst 1</b>	$q_4$		<b>iload 1</b> / <b>irem</b>	$q_4$
	<b>iconst 1</b> / <b>iconst 2</b>	$q_2$		<b>iconst 1</b> / <b>iload 2</b>	$q_2$
	<b>iconst 2</b> / <b>iload 1</b>	$q_0$		<b>iconst 2</b> / <b>iload 1</b>	$q_3$
	<b>iadd</b> / <b>iload 2</b>	$q_6$		<b>iadd</b> / <b>iconst 2</b>	$q_6$
	<b>iload 2</b> / <b>irem</b>	$q_7$		<b>iload 2</b> / <b>iadd</b>	$q_5$
<b>irem</b> / <b>iadd</b>	$q_5$	<b>irem</b> / <b>iconst 1</b>	$q_7$		
$q_2$	<b>goto</b> / <b>if icmpne</b>	$q_7$	$q_6$	<b>goto</b> / <b>if icmpne</b>	$q_2$
	<b>if icmpne</b> / <b>goto</b>	$q_0$		<b>if icmpne</b> / <b>goto</b>	$q_5$
	<b>iload 1</b> / <b>iload 2</b>	$q_6$		<b>iload 1</b> / <b>iload 1</b>	$q_4$
	<b>iconst 1</b> / <b>iconst 1</b>	$q_5$		<b>iconst 1</b> / <b>iconst 1</b>	$q_1$
	<b>iconst 2</b> / <b>iconst 2</b>	$q_4$		<b>iconst 2</b> / <b>iconst 2</b>	$q_6$
	<b>iadd</b> / <b>iadd</b>	$q_3$		<b>iadd</b> / <b>iadd</b>	$q_7$
	<b>iload 2</b> / <b>iload 1</b>	$q_1$		<b>iload 2</b> / <b>irem</b>	$q_0$
<b>irem</b> / <b>irem</b>	$q_2$	<b>irem</b> / <b>iload 2</b>	$q_3$		
$q_3$	<b>goto</b> / <b>goto</b>	$q_3$	$q_7$	<b>goto</b> / <b>goto</b>	$q_6$
	<b>if icmpne</b> / <b>if icmpne</b>	$q_6$		<b>if icmpne</b> / <b>if icmpne</b>	$q_2$
	<b>iload 1</b> / <b>iconst 1</b>	$q_2$		<b>iload 1</b> / <b>iconst 1</b>	$q_5$
	<b>iconst 1</b> / <b>iadd</b>	$q_5$		<b>iconst 1</b> / <b>iadd</b>	$q_3$
	<b>iconst 2</b> / <b>irem</b>	$q_1$		<b>iconst 2</b> / <b>iload 2</b>	$q_4$
	<b>iadd</b> / <b>iload 2</b>	$q_7$		<b>iadd</b> / <b>irem</b>	$q_1$
	<b>iload 2</b> / <b>iconst 2</b>	$q_4$		<b>iload 2</b> / <b>iload 1</b>	$q_7$
<b>irem</b> / <b>iload 1</b>	$q_0$	<b>irem</b> / <b>iconst 2</b>	$q_0$		