IEICE TRANS. FUNDAMENTALS, VOL.E88–A, NO.1 JANUARY 2005

PAPER Special Section on Cryptography and Information Security

Tamper-Resistant Software System Based on a Finite State Machine

Akito MONDEN[†*a)], ***Member***, **Antoine MONSIFROT**[†b)], ***and*** **Clark THOMBORSON**[†c)], ***Nonmembers***

SUMMARY Many computer systems are designed to make it easy for end-users to install and update software. An undesirable side effect, from the perspective of many software producers, is that hostile end-users may analyze or tamper with the software being installed or updated. This paper proposes a way to avoid the side effect without affecting the ease of installation and updating. We construct a computer system M with the following properties: 1) the end-user may install a program P in any conveniently accessible area of M; 2) the program P contains encoded instructions whose semantics are obscure and difficult to understand; and 3) an internal interpreter W, embedded in a non-accessible area of M, interprets the obfuscated instructions without revealing their semantics. Our W is a finite state machine (FSM) which gives context-dependent semantics and operand syntax to the encoded instructions in P; thus, attempts to statically analyze the relation between instructions and their semantics will not succeed. We present a systematic method for constructing a P whose instruction stream is always interpreted correctly regardless of its input, even though changes in input will (in general) affect the execution sequence of instructions in P. Our framework is easily applied to conventional computer systems by adding a FSM unit to a virtual machine or a reconfigurable processor.

key words: *software security, software protection, obfuscation, encryption*

1. Introduction

Security is an overarching problem for today's computer systems including personal computers, their peripherals, consumer electric devices, and any other machinery that contains software programs. Some systems administrators, and some software suppliers, require assurance that end-users will not analyze or tamper with protected programs or data. For example, a typical software digital rights management (DRM) system is designed to run in a "hostile" environment where the end-user is not fully trusted by the supplier of the content whose rights are being managed. Typically, these DRM systems contain cryptographic keys and algorithms that need to be kept secret [1]. There is, however, no known method for completely concealing these keys and algorithms from a determined attacker. For example, the keys for the CSS encryption standard for DVD media content were revealed by a "crack" in 1999. As a result, programs which subvert DVD copy protection are now widely distributed through the Internet [2]. Embedded software in consumer electric devices, e.g., mobile phones and set-top boxes, also needs to be protected since these devices are also susceptible to attacks by hostile users [3]. However, it seems impossible to completely prohibit end-user access to the software implementation, without also making it impossible to update this software to patch a "bug" or add a "feature."

In order to hide secrets in software implementation, software obfuscation techniques have been proposed [4]–[6]. Software obfuscations transform a program so that it is more difficult to understand, yet is functionally equivalent to the original program. Theoreticians have proved that "perfect obfuscation" is feasible on some programs but not for all programs, under a theoretically tractable but still interesting definition of "obfuscation" [7], [8]. However, it seems intuitively clear that a clever and well-equipped attacker would eventually understand any desired aspect of any program, no matter how well it is obfuscated, because any obfuscated program still contains all its original semantics. Thus, to maximise the security of a program, the widest possible variety of complementary obfuscation techniques should be used, for this will dissuade the widest possible range of attackers.

Instead of obfuscating the program itself, this paper presents a concept for obfuscating the program interpretation [9], [10]. 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 [11], [12]; however, in the randomization approach, the interpretation itself is not obscure because randomized instructions still have a one-to-one map to their semantics. On the other hand, our aim is to provide a dynamic map between instructions and their semantics.

In this paper we describe enhancements to our recently proposed framework for constructing an interpreter W, which carries out obfuscated interpretations for a given program P [9]. Here P is a translated version of an original program P_0 written in a common programming language (such as Java bytecode and x86 assembly.) The obfuscated interpretation means that an interpretation W for a given instruction c is not fixed; specifically, the interpretation $W(c)$ is determined not only by c itself but also by previous instructions input to W (Figure 1).

In order to realize an obfuscated interpretation in W, we employ a FSM that takes as input an instruction c where

Manuscript received March 12, 2004.
Manuscript revised June 18, 2004.
Final manuscript received September 8, 2004.

[†]The authors are with the Department of Computer Science, The University of Auckland, New Zealand.
[*]Presently, with Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-0192 Japan.
a) E-mail: akito-m@is.naist.jp
b) E-mail: antoine@cs.auckland.ac.nz
c) E-mail: cthombor@cs.auckland.ac.nz

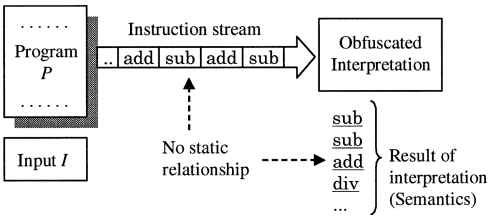

Fig. 1 Concept of obfuscated interpretation.

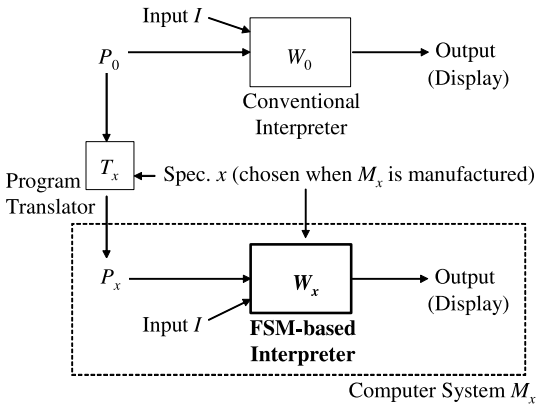

Fig. 2 Framework for obfuscated interpretation.

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 with respect to previous inputs. Such a 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 by an updated program.

In our original proposal [9], we had required the opcode encoding to preserve the number and type of the operands. In this paper we demonstrate how to build a FSM without this restriction. This increases the range of possibilities from which the FSM W is chosen, which has an effect analogous to increasing the "key length" of a cryptographic cipher. That is, the proposal in this paper is more resilient to brute-force enumerative ("naïve key-search") attacks. This paper also extends its predecessor by demonstrating an example in x86 assembly code rather than in Java bytecode; this extension required us to add a dead-register analysis to our process for the obfuscation of code by interpretation. Note that a full but older version of our proposal has been described in our technical report [10].

In some sense, the mechanism of our obfuscated interpretation is a kind of stream cipher where a ciphered bit sequence is decoded one bit (or one byte) at a time dependent on its context [13]; however, conventional stream ciphers cannot 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 (or a special instruction for FSM control) into P to force expedient state transitions in W so that P is always interpreted correctly regardless of its input.

The rest of this paper is organized as follows. In Section 2, a framework for obfuscated interpretation is described. Section 3 shows a case study of obfuscated interpretation. Section 4 discusses several attacks and defences. Section 5 compares our approach with conventional software encryption approaches. Finally, Section 6 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. We assume that 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 "translated program" containing encoded 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.
- 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 encoded instructions (inputs of the FSM) and their semantics (outputs of the FSM). This x is used in both a FSM-based interpreter W_x and a program translator T_x.
- W_x: is a FSM-based interpreter that can evaluate encoded 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 translated 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 from reading the implementation of Wx. 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 knows 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.

2.2 FSM-Based Interpreter

2.2.1 Design Types

We identify five 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 call Ins_{P_0} a set of "cleartext instruction" and Ins_{P_x} a "encoded instruction." We define five types of designs. Note: in our original proposal [9] we had not defined "Type 2.5."

(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 static semantics of P_x may be incorrect (e.g., if L_{P_0} is Java bytecode, 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, which will increase the size of the search space in a brute-force attack. However, there is a security tradeoff, for an attacker might use a static syntax checker to quickly discard many possible de-obfuscations in a brute-force search. So we do not, as yet, know which type of obfuscation will give the most security in practice.

(Type 2.5) L_{P_x} has different operand syntax to L_{P_0}; individual opcodes in P_0 are translated into opcodes in P_x with the same number of bytes; and the opcode sets and encodings in Ins_{P_0} and Ins_{P_x} are identical. Because the type and number of operands (and their specifiers) associated with each opcode may differ from L_{P_0}, P_x is generally not a valid program in L_{P_0}. The number of different FSMs that could be used to interpret P_x is larger than in Type 2.

(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 2.5.

(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 an Ins_{P_0} 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.

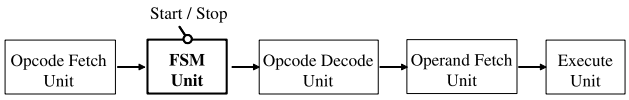

Fig. 3 Pipelined stages of FSM-based interpreter.

In the rest of this paper, we focus on Type 2.5 designs.

2.2.2 Architecture

Figure 3 shows a suitable architecture for a FSM-based interpreter, characterized by simplified pipelined stages of interpretation. In this paper we focus on the opcodes to be translated in the FSM. In Type 2.5 design, the FSM-based interpreter is augmented by an additional pipeline stage, called a FSM unit, which translates a "Type-2.5" encoded opcode into a cleartext opcode, passing it to a conventional opcode decode unit. Then, the cleartext opcode is (syntactically) decoded, and the number of operands to be fetched is determined. After the required operands are fetched in an operand fetch unit, the instruction is executed in an execute unit. This architecture is applicable to many present Java Virtual Machines (JVMs) and reconfigurable processors.

The FSM unit has a switch to start/stop the obfuscated interpretation to enable us to run both an ordinary program and a translated program on the same interpreter. If the FSM unit is stopped, the interpreter works as an ordinary one, and if it is started, 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 instruction.

2.2.3 FSM Unit

The FSM unit (denoted as w_x) is a DFA (Deterministic Finite Automaton) defined by 6-tuple $(Q, \Sigma, \Psi, \Delta, \Lambda, q0)$ where

$Q = \{q_0, q_1, ..., q_{n-1}\}$ is the states in the FSM.
$\Sigma = \{c_0, c_1, ..., c_{n-1}\}$ is the input (encoded opcodes).
$\Psi = \{\underline{c_0}, \underline{c_1}, ..., \underline{c_{n-1}}\}$ is the output (cleartext opcodes).
$\delta_i : \Sigma \rightarrow \underline{Q}$ is the next-state function for state q_i.
$\Delta = \{\delta_0, \delta_1, ..., \delta_{n-1}\}$ is the n-tuple of all next-state functions.
$\lambda_i : \Sigma \rightarrow \Psi$ is the output function for state q_i.
$\Lambda = \{\lambda_0, \lambda_1, ..., \lambda_{n-1}\}$ is the n-tuple of all output functions.
$q_0 \in Q$ is the starting state of the FSM.

In Type 2.5 design, the instruction set for P_x is the same as that for P_0, i.e., $Ins_{P_x} = Ins_{P_0}$. We assume $Ins_{P_x} = \Sigma \cup O$ where elements $c_i \in \Sigma$ are encoded instructions, and $o_i \in O$ are non-encoded instructions. This means that P_x contains both c_i and o_i, and, if the FSM unit recognizes a $c_i \in \Sigma$ as input, then it is translated into a cleartext opcode by the FSM and it is passed to the execute unit, otherwise an input $o_i \in O$ will not be translated (directly passed to the execute unit). In our Type-2.5 design, each underlined symbol $\underline{c_i}$ in Ψ denotes the normal (cleartext) 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 $c_0, c_1, ..., c_{b-1}$ are in the first class C_1 (of branching opcodes including non-conditional jump) and the remaining symbols $c_b, c_{b+1}, ..., c_{n-1}$ are in the second class C_2(of non-branching opcodes). The FSM design has the following constraints.

1. Each $\delta_i : \Sigma \to Q$ is a bijection; we write its inverse as $\delta_i^{-1} : Q \to \Sigma$.
2. Each $\lambda_i : \Sigma \to \Psi$ is a bijection, defining $\lambda_i^{-1} : \Psi \to \Sigma$. Note that the opcode sets in Σ and Ψ are identical in our Type-2.5 FSM design.
3. For all i and j, the length of the cleartext opcode $\lambda_i(c_j)$ is the same as the encoded opcode c_j so that the opcode fetch unit can correctly fetch encoded opcodes.
4. For all pairs of states q_i, q_k there exists a "dummy instruction sequence" d_j with the following three properties. First, d_j is a short sequence of instructions containing exactly one encoded instruction. Second, an FSM initially in state q_i will be in state q_k after it takes d_j as input. Third, $\underline{d_j}$ (cleartext dummy sequence) has no effective functionality. Thus $\underline{d_j}$ 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 δ_i is a bijection. The encoded instruction in $\underline{d_j}$ 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 q_k.)

Figure 4 shows a simple example of w_x where

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

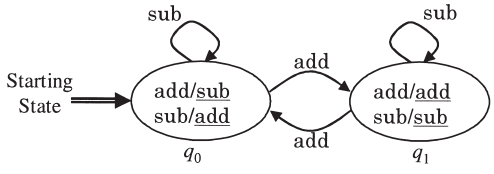

Fig. 4 Example of FSM w_x.

Fig. 5 Example of instruction stream interpretation.

This w_x takes an encoded opcode $c_i \in \{add, sub\}$ as an input, translates it into its semantics (cleartext opcode) $\underline{c_i} \in \{\underline{add}, \underline{sub}\}$, and outputs $\underline{c_i}$. Figure 5 shows an example of interpretation for an instruction stream given by this w_x. Obviously, even this simple FSM has the ability to conduct an obfuscated interpretation. As shown in Figure 5, the opcode "add" is interpreted as either add or sub according to its context.

2.2.4 Program Translator

In order to develop a program executable on the FSM-based interpreter, a program translator $T_x : P_0 \to P_x$ is indispensable. Since the interpreter w_x translates encoded instructions in P_x into cleartext instructions in P_0, an inverse interpreter of w_x (denoted by ${w_x}^{-1}$) is useful for translating instructions in P_0 into ones in P_x. However, building ${w_x}^{-1}$ is not sufficient to process the translation $T_x : P_0 \to P_x$. Let us assume we have w_x of Figure 4, and P_0 of Figure 6 that computes a summation $p := 1 + 2 + 3 + ... + n$. 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 in Section 2.2.3, 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 the program semantics.

From the above discussion, we first build an inverse interpreter ${w_x}^{-1}$, then we use this ${w_x}^{-1}$ and dummy instruction sequences to translate P_0 into P_x. Our ${w_x}^{-1}$ is the DFA defined by 6-tuples $(Q', \Sigma', \Psi', \Delta', \Lambda', q_0)$ where

$Q' = Q = \{q_0, q_1, ..., q_{n-1}\}$ is the FSM's states.
$\Sigma' = \Psi = \{\underline{c_0}, \underline{c_1}, ..., \underline{c_{n-1}}\}$ is its input.
$\Psi' = \Sigma = \{c_0, c_1, ..., c_{n-1}\}$ is its output.
$\delta'_i : \Sigma' \to Q'$ is its next-state function for state q_i, where $\delta'_i(\underline{c_j})$ has the value $\delta_i(\lambda_i^{-1}(\underline{c_j}))$ for all i, j.
$\Delta' = \{\delta'_0, \delta'_1, ..., \delta'_{n-1}\}$ is the n-tuple of all next-state functions.
$\lambda'_i : \Sigma' \to \Psi'$ is the output function for state q_i, where each $\lambda'_i : \Sigma' \to \Psi'$ has the value $\lambda_i^{-1}(\underline{c_j})$ for all i, j.
$\Lambda' = \{\lambda'_0, \lambda'_1, ..., \lambda'_{n-1}\}$ is the n-tuple of all output functions.
$q_0 \in Q'$ is the starting state of this FSM.

Figure 7 shows an example of the ${w_x}^{-1}$ corresponding to the w_x of Figure 4. As shown in Figure 7, ${w_x}^{-1}$ has the same number of states and transitions as w_x.

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

Fig. 6 Example of P_0.

Next, we give a procedure for the translation $T_x : P_0 \rightarrow P_x$. Appendix 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.

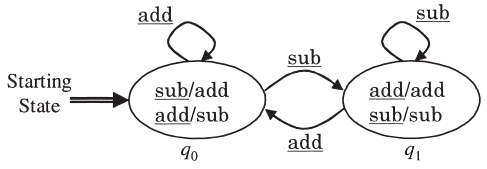

Fig. 7 Example of ${w_x}^{-1}$.

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

Fig. 8 Example of translated 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 8 shows an example of P_x translated from P_0 of Figure 6. 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_x comes to q_0 every time the execution reaches the entry point of the loop.

3. Case Study

3.1 Program Translation

In this section we explain a more complex example of program translation $T_x : P_0 \rightarrow P_x$ using the inverse interpreter ${w_x}^{-1}$ given in Table 1. This ${w_x}^{-1}$ is designed for programs written in an Intel x86 instruction set. We use the AT&T syntax (GNU assembler format) to write assembly codes in L_{P_0} and L_{P_x}. Our sample ${w_x}^{-1}$ has eight states $Q' = \{q_0, q_1, ...q_7\}$ with q_0 a starting state, and has eight types of instructions $\Sigma' = \{$jmp, jne, pushl, decl, movl$_{89}$, subl, movl$_{8B}$, leal$\}$. Here, "movl$_{89}$" indicates MOV instructions whose opcode byte is "89," and "movl$_{8B}$" indicates the "8B" variant of this opcode. Two instructions (jmp and jne)

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

State	Input / Output			Transition	State	Input / Output			Transition
q_0	EB jmp	/	75 jne	q_4	q_4	EB jmp	/	75 jne	q_6
	75 jne	/	EB jmp	q_7		75 jne	/	EB jmp	q_4
	55 pushl %ebp	/	4D decl %ebp	q_2		55 pushl %ebp	/	89 movl$_{89}$	q_2
	4D decl %ebp	/	55 pushl %ebp	q_5		4D decl %ebp	/	4D decl %ebp	q_1
	89 movl$_{89}$	/	89 movl$_{89}$	q_1		89 movl$_{89}$	/	29 subl	q_3
	29 subl	/	29 subl	q_6		29 subl	/	8B movl$_{8B}$	q_0
	8B movl$_{8B}$	/	8D leal	q_3		8B movl$_{8B}$	/	8D leal	q_7
	8D leal	/	8B movl$_{8B}$	q_0		8D leal	/	55 pushl %ebp	q_5
q_1	EB jmp	/	75 jne	q_1	q_5	EB jmp	/	EB jmp	q_2
	75 jne	/	EB jmp	q_3		75 jne	/	75 jne	q_1
	55 pushl %ebp	/	8D leal	q_4		55 pushl %ebp	/	8B movl$_{8B}$	q_4
	4D decl %ebp	/	55 pushl %ebp	q_2		4D decl %ebp	/	55 pushl %ebp	q_0
	89 movl$_{89}$	/	8B movl$_{8B}$	q_0		89 movl$_{89}$	/	29 subl	q_3
	29 subl	/	29 subl	q_6		29 subl	/	89 movl$_{89}$	q_6
	8B movl$_{8B}$	/	89 movl$_{89}$	q_7		8B movl$_{8B}$	/	4D decl %ebp	q_5
	8D leal	/	4D decl %ebp	q_5		8D leal	/	8D leal	q_7
q_2	EB jmp	/	75 jne	q_7	q_6	EB jmp	/	75 jne	q_5
	75 jne	/	EB jmp	q_0		75 jne	/	EB jmp	q_2
	55 pushl %ebp	/	29 subl	q_6		55 pushl %ebp	/	8D leal	q_4
	4D decl %ebp	/	8D leal	q_5		4D decl %ebp	/	89 movl$_{89}$	q_1
	89 movl$_{89}$	/	55 pushl %ebp	q_4		89 movl$_{89}$	/	8B movl$_{8B}$	q_6
	29 subl	/	8B movl$_{8B}$	q_3		29 subl	/	4D decl %ebp	q_7
	8B movl$_{8B}$	/	89 movl$_{89}$	q_1		8B movl$_{8B}$	/	55 pushl %ebp	q_0
	8D leal	/	4D decl %ebp	q_2		8D leal	/	29 subl	q_3
q_3	EB jmp	/	EB jmp	q_3	q_7	EB jmp	/	75 jne	q_0
	75 jne	/	75 jne	q_5		75 jne	/	EB jmp	q_6
	55 pushl %ebp	/	4D decl %ebp	q_2		55 pushl %ebp	/	4D decl %ebp	q_5
	4D decl %ebp	/	89 movl$_{89}$	q_6		4D decl %ebp	/	89 movl$_{89}$	q_3
	89 movl$_{89}$	/	29 subl	q_1		89 movl$_{89}$	/	55 pushl %ebp	q_4
	29 subl	/	55 pushl %ebp	q_7		29 subl	/	29 subl	q_1
	8B movl$_{8B}$	/	8D leal	q_4		8B movl$_{8B}$	/	8D leal	q_7
	8D leal	/	8B movl$_{8B}$	q_0		8D leal	/	8B movl$_{8B}$	q_2

are branching instructions (class C_1). The other six are non-branching instructions (class C_2). For each instruction in Table 2, a binary (hexadecimal) representation of the opcode is shown.

Table 2 shows sequences of dummy instructions $\underline{d_i}$ for each $\underline{c_i} \in \Sigma'$. 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. Some of these d_i modify registers, and these must be "dead registers" to define our desired no-op function. Dead registers can be detected easily by static analysis of P_0 [14]. For increased security, in a future implementation, we would make it difficult for an attacker to discover that these registers are in fact dead; for example, we could add their values to the parameter list of a function so that they would be pushed and popped during function calls and returns.

Table 2 Example of dummy instruction sequences.

opcode $\underline{c_i}$	dummy sequence $\underline{d_i}$
jmp	jmp .Lx any instructions .Lx
jne	jne .Lx addl $0,%eax .Lx
pushl	pushl %eax pop %eax
decl	decl %eax incl %eax
movl_{89}	movl_{89} %eax,%ebx†
movl	movl $0, %eax subl %ebx,%eax
movl_{8B}	movl_{8B} -4(ebp),%eax†
leal	leal -4(ebp),%eax†

† these registers must be dead

The target P_0, which is to be translated, is shown in Figure 9. This P_0 is a x86 assembly program, compiled by gcc from the C source program. This P_0 computes a summation of odd numbers $p := 1 + 3 + 5 + ... + n$. Figure 10 shows P_x corresponding to this P_0. In Figure 9, numbers described in the leftmost column indicate line numbers, and their corresponding lines are described in Figure 10 as well. The second column in Figure 9 describes the state of w^{-1} in which each instruction is interpreted. Due to limited space, we have not included a detailed explanation of our translation process in this paper. However, a full explanation of a sample translation of a Java bytecode program for a Type-2 interpreter is shown in our previous paper [9].

On the basis of our case study, we can estimate the overheads of our protection mechanism on a typical x86 code. Such codes have approximately one branch for every seven instructions, and we will introduce approximately 1.5 dummy instructions for each branch. Thus the code-

```
1   q0  pushl %ebp
2   q2  movl89 %esp, %ebp
3   q4  subl $8, %esp
4   q4  movlC7 $0, -4(%ebp)
5   q4  movlC7 $1, -8(%ebp)
6   q4  .L3:
7   q4  movl8B -8(%ebp), %eax
8   q7  cmpl 8(%ebp), %eax
9   q7  jle .L6
10  q7  jmp .L4
11  q7  .L6:
12  q7  movl8B -8(%ebp), %edx
13  q7  movl89 %edx, %eax
14  q4  sarl $31, %eax
15  q4  shrl $31, %eax
16  q4  leal (%eax,%edx), %eax
17  q4  sarl $1, %eax
18  q4  sall $1, %eax
19  q4  subl %eax, %edx
20  q0  movl89 %edx, %eax
21  q1  cmpl $1, %eax
22  q1  jne .L5
23  q3  movl8B -8(%ebp), %edx
24  q4  leal -4(%ebp), %eax
25  q5  addl %edx, (%eax)
    q5
26  q3  .L5:
27  q3  leal -8(%ebp), %eax
28  q0  incl (%eax)
29  q0  jmp .L3
30  q0  .L4:
31  q0  movl8B -4(%ebp), %eax
32  q3  leave
33  q3  ret
```

Fig. 9 P_0 in GNU assembler format.

```
1   decl %ebp
2   pushl %esp, %ebp
3   subl $8, %esp
4   movlC7 $0, -4(%ebp)
5   movlC7 $1, -8(%ebp)
6   .L3:
7   leal -8(%ebp), %eax
8   cmpl 8(%ebp), %eax
9   jle .L6
10  jne .L4
11  .L6:
12  leal -8(%ebp), %edx
13  pushl %edx, %eax
14  sarl $31, %eax
15  shrl $31, %eax
16  leal (%eax,%edx), %eax
17  sarl $1, %eax
18  sall $1, %eax
19  movlC7 %eax, %edx
20  movl89 %edx, %eax
21  cmpl $1, %eax
22  jmp .L5
23  leal -8(%ebp), %edx
24  pushl -4(%ebp), %eax
25  addl %edx, (%eax)
    subl %eax ,%ebx    ; dummy q5 → q3
26  .L5:
27  movlC7 -8(%ebp), %eax
28  incl (%eax)
29  jne .L3
30  .L4:
31  leal -4(%ebp), %eax
32  leave
33  ret
```

Fig. 10 P_x in GNU assembler format.

size of a translated program will increase modestly: by approximately 20%. Because our dummy instructions have no data or control hazards (i.e., they will cause neither cache misses nor branch mispredictions), they will execute more efficiently than an average untranslated instruction. We conclude that the runtime and space overhead of our translation process will be at most 20% on a typical x86 code.

3.2 Obscurity of Translated Program

The program P_x obtained by our translation has some fundamental characteristics to make itself obscure. Below we describe the characteristics of P_x in Figure 10 compared with P_0 in Figure 9.

1. As described in 2.2.1, P_x uses the opcodes of an original x86 assembly language L_{P_0}, but it is not itself a valid x86 program since the operand signatures in P_x are not all correct in L_{P_0}. For example, in line 2 of Figure 10, the "pushl" opcode requires one operand in L_{P_0}, however, it has two operands in P_x. This indicates P_x cannot be parsed accurately by a disassembler for L_{P_x} into instructions.
2. Instructions in P_x do not have static binding to their semantics. For example, "pushl" in line 2 is interpreted as "movl89" via w_x (see the same line in Figure 9), but in line 24, it is interpreted as "leal." Note that dummy instructions, for example the one between line 25 and 26, also have non static semantics, so they are not statically recognizable as dummy instructions.
3. The control flow of P_0 is not apparently preserved in P_x, i.e., if P_x were executed without translation "just as it appears," it would take different branches than P_0. For example, the conditional jump "jne" in line 10 is actually an unconditional JMP.

4. Security Analysis

In this section, we analyze the security of our scheme against adversaries with varying resources, knowledge, and persistence. However, due to limited space, here we describe the summary of our analysis. For more detailed analysis, please see our technical report [10].

Generally speaking, our security objective is to prevent an adversary from understanding (and also tampering with) the protected software. However, it is difficult to prevent all the imaginable attacks. For example, it is difficult to completely prevent the making of small alterations in a program to defeat a license check [15]. In this paper, we limit our objective to prevent the adversary from understanding the protected software sufficiently well to make large-scale alterations 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. Many cryptograhic systems require this kind of tamper resistance [1]. In this paper, we consider that if an attacker could discover the specification of the FSM, then, the attacker is able to understand P_x sufficiently well to modify P_x. The immediate goal of this section is to analyze the difficulty of discovering the specification of the FSM.

We characterize an adversary's knowledge and resources along three dimensions (labeled A, B, C), as listed below. To simplify our analyses, we consider only adversaries who are equal on all dimensions. An adversary that is at level-0 is a naive end-user. Our level-1 adversary is an end-user with very limited technical skill and ability. Our level-2 adversary has a debugger and good technical skills.

(A) FSM Interpretation

0. The level-0 adversary has 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. The adversary has an algorithmic understanding of the principles of FSM-based interpretation, as described in this article.
2. The adversary has a debugger with "breakpoint" functionality, attached to a translated 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.

(B) System 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. This means, the level-2 adversary can observe (a few) cleartext instructions while the level-1 adversary can only guess at them from the audio-visual outputs.

(C) System 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. Note that the level-2 adversary can interrupt the execution of P_x by using a debugger before injecting instructions to W_x, while the level-1 adversary can only run a modified P_x from the beginning.

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

We turn to the level-1 attacks. A cryptographically skilled adversary with knowledge of programming language semantics and our FSM algorithm would, we imagine, attempt a brute-force attack by code injection.

A plausible first step in such an attack would be to build a table of "dummy instruction sequences" d_j 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-encoded instructions, a single encoded instruction, and a short (possibly empty) suffix of non-encoded instructions. Algorithm T_x will place a dummy instruction sequence at the end of the 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. Therefore, the suffixes will be recognizable as 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 cleartext opcodes) in our Type 2.5 FSM design, although the adversary will certainly make a number of mistakes in the recognition of branching opcodes wherever instruction boundaries are obscure in the encoded code due to the differing operand syntax in L_{P_0} and L_{P_x}. For example, a 0xEB byte in the encoded sequence is reasonably likely to be an encoded branch opcode but it may also be an operand.

The adversary might examine the $O(n^2)$ loops to be reasonably certain of having discovered all suffixes, so hypothesizing d_j may take days, but not months, if $n < 256$. Here, n denotes the number of states in the FSM. The prefixes can be recognized as 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.

The next step in this brute-force code-injection attack is to find a sensitive loop, that is, a loop in which the modification of a single opcode will visibly and quickly affect program operation (a level-1 observation). We imagine that an adversary will quickly discover such a loop, if they inject arbitrary opcodes into randomly chosen loops. The attacker should also insert a short no-op sequence at this position, to confirm that program correctness is not hypersensitive to loop timing.

The third step is to construct a list of possible (hypothesized) encodings for each dummy sequence d_j identified in the first step. Each entry d_{jk} in the list is a modified version of d_j, where the encoded opcode c_j (the translated opcode in d_j) is replaced with another opcode c_k.

The fourth step is to choose one pair d_{ij}, d_{kz} of the (hypothesized) encoded dummy sequences for insertion at this point in the program. A small fraction of these pairs (about $1/n^2 = 1/40000$ if there are $n = 200$ encoded opcodes) will not affect program correctness. An adversary can thus "crack" one of the n^2 transitions in the FSM by making $O(n^2)$ observations and controls on a modified P'_x for their machine M_x. After n^2 such discoveries, the FSM is completely cracked. This attack will take $O(n^4)$ observations and controls, where each observation (of whether or not program correctness has been impaired) can be accomplished in a few seconds. This is thus a (barely) feasible attack: a dedicated adversary will crack a single FSM in a few months.

Faster attacks may exist, of course. As in any cryptographic security analysis, we can only place a lower bound on an attacker's "crack time" if we are able to enumerate, and then analyze, all possible attacks. This is generally not feasible; there is always the possibility of an attacker finding a previously unknown attack that will run much more quickly than any known attack. However, our system will be secure against level-1 attack unless someone, at some time, devises a more efficient means of attack than the brute-force code-injection attack described above.

If the brute-force attack described above is considered to be a significant security risk in a certain application, then we can easily modify our translation process to increase the adversary's search space. We could use multiple "dummy sequences" for each instruction. We could randomize the locations in which we insert "dummy sequences" (our translation algorithm T_x could insert a dummy sequence at any point in the straight-line code leading up to a branchpoint in P_0). We could use a Type-2 or Type-2.5 design without a partition between branching and non-branching opcodes. We could use a Type-3 FSM to make it harder for the attacker to recognize no-op suffixes and prefixes. Finally, we could use a Type-4 FSM to increase n. We intend to explore these options in future work.

We now briefly consider level-2 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. Thus a single observation and control can be targeted at each of the FSM's n^2 state transitions, so a level-2 attacker will "crack" a FSM in $O(n^2)$ observations and controls, each taking a few seconds.

When our design is applied to typical instruction sets, for example Java bytecode or x86 machine code, then $n \approx 200$. In this case, the level-2 attack described above is quite potent. We therefore recommend, on the basis of our security analysis, that our FSM should be implemented in physically secure hardware on the CPU chip, with mechanisms to obstruct logic analyzers (i.e., [14]).

We have considered more complex FSM designs, with more than one state per opcode, however, we believe all such designs would not be feasible for the foreseeable fu-

ture. When $n > 256$, the RAM storage for the FSM's state-transition table becomes larger than $n^2 = 65,536$ bytes. This will be expensive to fabricate on a CPU chip, and if it is placed off-chip then it will be difficult to secure against direct observation by the attacker.

We close our security analysis with a caution. Our translation system is essentially cryptographic in nature, so it should only be used to translate 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. Related Work

The most commonly proposed method for hiding interpretation is program encryption [16], [17]. Figure 11 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, $E(P_0)$ itself is not understandable to the user. Also, many different programs can run on a single system M_k, and they are easily updatable.

There are two types of architectures that can implement this approach: (1) R is the processor's ICache (instruction cache), e.g., XOM [18], AEGIS [19] and Cerium [20], and (2) a very large secure R is provided by the Operating System, e.g., Trusted Computing (also known as TCPA, Palladium and NGSCB) promoted by TCG [21]. In the former case, the ICache miss penalty is increased by the decryptor's latency. Since the long blocklengths of strong cryptography add considerable latancy, it is intolerable in many applications. Consider what happens on each unpredicted branch into a not-yet-decrypted block. All bytes in this block, prior to the branch target, must be fetched and decrypted before execution can recommence. By contrast, our FSM-based approach has dummy instructions inserted into cipher text (i.e., translated program) controlling the decryptor (i.e., FSM-based interpreter) so that it allows something akin to random-access to all branch targets (but only from the source of the branch) without losing synchronization. We expect that opcode translation $c_i \rightarrow \underline{c_i}$ by the FSM would take only 1 clock cycle, which would not not cause significant performance degradation in either the simplified processor pipeline shown in Figure 3 or the deeper pipelines of modern microprocessors.

In the latter case of a large secure RAM, there is no additional ICache miss penalty. However, there is a startup penalty since the entire program must be decrypted before execution, while our FSM can start immediately. Another concern is that, in this architecture, we need to trust the OS. However, since it is difficult to guarantee that the OS has no bugs or security holes, our interpretation obfuscation can add another layer of protection. In addition, since

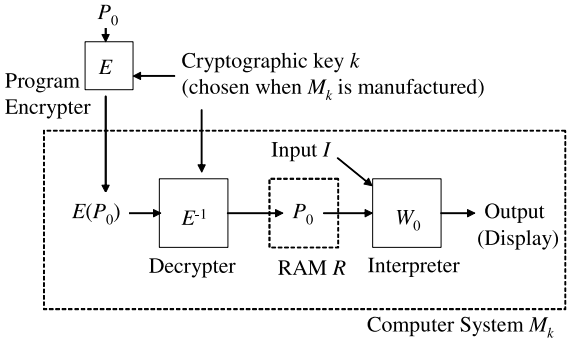

Fig. 11 Basic approach for program encryption.

building a secure RAM may increase the cost, our approach may be an attractive approach. In our approach, building the FSM-based interpreter W_x is easily realized by adding a small FSM unit to current hardware-based virtual machines (such as [22], [23]) and reconfigurable processors.

Program encryption may also be employed at the bus level of a CPU architecture, rather than at the memory level [24]. In this case, the address of the memory location may be used as part of the cryptographic key. Typically the blocklength of the cypher is short: for efficiency reasons it may be a single 8-bit memory word. (Note that if long blocklengths and strong cryptographic methods were used, then the bus-encryption method would incur the latency penalties noted in our discussion of Figure 11.) Because of its short blocklength, the bus-encryption method is highly susceptible to brute-force chosen cyphertext attacks. A level-1 attacker, in our taxonomy of Section 4, may place an arbitrary cyphertext in memory and observe system behaviour. Kuhn has used this technique in a level-2 attack on the bus-encryption security microcontroller DS5002FP. His brute-force search revealed a short cyphertext sequence that had the effect of block-translating all encrypted bytes in the system memory [24]. Kuhn's attack is not feasible against our method, because the CPU in our system has no instructions that would export decoded opcodes outside the CPU core.

We believe that the encryption of code, obfuscation of code and obfuscation of interpretation are not exclusive techniques, and indeed that all of these techniques can and should be used as complementary techniques in highly secure software systems. In our technical report [10], we discuss other techniques (e.g., [25]) that may be used to hide program interpretation.

6. Conclusion

In this paper we proposed a framework for obfuscating the program interpretation, which is more powerful than code obfuscation, and is faster and cheaper than encryption schemes. We defined a FSM-based interpreter w_x that provides 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 2.5" translation of an x86 assembly-language P_0 into an "x86-like" P_x showed that instructions in P_x have nonstatic semantics, i.e., functionality is hidden from program users, yet P_x is still functionally equivalent to P_0. Our preliminary security analysis highlighted some areas where our design could be improved, and we conclude that our design should only be used to translate long programs that have been "randomized" (i.e., obfuscated) before they are translated. In the future, we should develop a method for operand encoding since operand values leak information about opcodes. We will also develop detailed designs for interpreters of Type 1, 3 and 4, and we intend to clarify their advantages and shortcomings.

References

[1] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, "A white-box DES implementation for DRM applications," Proc. 2nd ACM Workshop on Digital Rights Management (DRM2002), Washington DC, USA, 2002.
[2] A. Patrizio, "Why the DVD hack was a cinch," Wired News, 1999. http://www.wired.com/news/technology/0,1282,32263,00.html
[3] The U.K. Parliament, "The mobile telephones (re-programming) bill," House of Commons Library Research Paper 02-47, 2002. http://www.parliament.uk/commons/lib/research/rp2002/rp02-047.pdf
[4] F.B. Cohen, "Operating system protection through program evolution," Comput. Secur., vol.12, no.6, pp.565–584, Elsevier Science, 1993.
[5] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—Tools for software protection," IEEE Trans. Softw. Eng., vol.28, no.8, pp.735–746, 2002.
[6] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," IEICE Trans. Fundamentals, vol.E86-A, no.1, pp.176–186, Jan. 2003.
[7] B. Barak, P. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," Lecture Notes in Computer Science, vol.2139, pp.1–18, Springer-Verlag, 2001.
[8] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," Eurocrypt 2004, Lecture Notes in Computer Science, vol.3027, pp.20–39, Springer-Verlag, May 2004.
[9] A. Monden, A. Monsifrot, and C. Thomborson, "A framework for obfuscated interpretation," Australasian Information Security Workshop (AISW2004), ed. P. Montague and C. Steketee, ACS, CRPIT, vol.32, pp.7–16, 2004.
[10] A. Monden, A. Monsifrot, and C. Thomborson, "Obfuscated instructions for software protection," Information Science Technical Report, NAIST-IS-TR2003013, Nara Institute of Science and Technology, 2003.
[11] E.G. Barrantes, D.H. Ackley, S. Forrest, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," Proc. 10th ACM Conference on Computer and Communications Security (CCS2003), pp.281–289, Washington DC, USA, 2003.
[12] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," Proc. 10th ACM Conference on Computer and Communications Security (CCS2003), pp.272–280, Washington DC, USA, 2003.
[13] M.J.B. Robshaw, "Stream ciphers," RSA Laboratories Technical Report TR-701, 1995.
[14] J. Irwin, D. Page, and N.P. Smart, "Instruction stream mutation for non-deterministic processors," Proc. 13th International Conference on Application-specific Systems, Architectures and Processors (ASAP2002), pp.286–295, 2002.
[15] M. LaDue, "The Maginot license: Failed approaches to licensing Java software over the Internet," 1997. http://www.geocities.com/securejavaapplets/maginot.html
[16] D.J. Albert and S.P. Morse, "Combatting software piracy by encryption and key management," Computer, vol.17, no.4, pp.68–73, 1984.
[17] A. Herzberg and S.S. Pinter, "Public protection of software," ACM Trans. Comput. Syst., vol.5, no.4, pp.371–393, 1987.
[18] D. Lie, C.A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," Proc. 19th ACM Symposium on Operating Systems Principles, Oct. 2003.
[19] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," Proc. 17th International Conference on Supercomputing (ICS2003), pp.160–171, June 2003.
[20] B. Chen and R. Morris, "Certifying program execution with secure processors," Proc. 9th Workshop on Hot Topics in Operating Systems, May 2003.
[21] The Trusted Computing Group. https://www.trustedcomputing group.org/
[22] TinyJ, Advancel Logic Corporation. http://www.advancel.com/products.htm
[23] Xpresso, Zucotto Wireless Inc. http://www.zucotto.com/home/
[24] M.G. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP," IEEE Trans. Comput., vol.47, no.10, pp.1153–1157, 1998.
[25] T. Maude and D. Maude, "Hardware protection against software piracy," Commun. ACM, vol.27, no.9, pp.950–959, 1984.

Appendix: Procedure for $T_x : P_0 \rightarrow P_x$

Let $state(k)$:= NULL for all line number k of P_0
Let $q_s := q_0$
Set PC to the entry point of P_0
loop:
  If PC = exit of P_0 then goto resume
  If $state(PC) \neq$ 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 { /* encoded 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-encoded instruction */
    Let $code_{P_x}(PC) := code_{P_0}(PC)$
  }
  If $code_{P_0}(PC)$ = branching instruction then {
    If $code_{P_0}(PC) \neq$ non-conditional jump then {
      Do $push(PC_{false})$ where PC_{false} is a line number of next instruction in $false$ branch
      Let $state(PC_{false}) = q_s$
    }
    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_{prev} := previous value of PC
  If $code_{P_0}(PC_{prev})$ = 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 $\overline{d_i} := \lambda_s^{-1}(\underline{d_i})$
    Insert d_i into $\overline{P_x}$ right after the line number = PC_{prev}
  }else{
    Choose k that satisfies $\delta'_k(code_{P_0}(PC_{prev})) = state(PC)$
    Choose $\underline{c_i} \in \Sigma'$ that satisfies $\delta'_{state(PC_{prev})}(\underline{c_i}) = q_k$
    Let $\underline{d_i}$:= a sequence of dummy instructions for $\underline{c_i}$
    Let $\overline{d_i} := \lambda_s^{-1}(\underline{d_i})$
    Insert d_i into $\overline{P_x}$ at the line number = PC_{prev}
    $state(PC_{prev}) = q_k$
  }
  return

Akito Monden received the B.E. degree (1994) in electrical engineering from Nagoya University, Japan, and the M.E. degree (1996) and D.E. degree (1998) in information science from Nara Institute of Science and Technology, Japan. He was honorary research fellow at the University of Auckland, New Zealand, from June 2003 to March 2004. He is currently Associate Professor at Nara Institute of Science and Technology. His research interests include software security, software measurement, and human-computer interaction. He is a member of the IEEE, ACM, IPSJ, and JSSST.

Antoine Monsifrot received the Bachelor degree (1998) in computer science from University of Rennes, France, and the Ph.D. degree (2002) in computer science from IRISA (National Institute for Research in Computer Science), France. He is currently Research Fellow at the University of Auckland, New Zealand. His research interests include program transformation, code optimization, and software security.

Clark Thomborson received the Ph.D. degree (1980) in computer science, under his birth name Clark Thompson, from Carnegie-Mellon University, Pennsylvania. He emigrated to New Zealand in 1996, to take up his current position as Professor of Computer Science at the University of Auckland, New Zealand. He has published more than 70 refereed papers on topics in software security, computer systems performance analysis, VLSI algorithms, data compression, and connection networks. He is a senior member of the IEEE and a member of the ACM and RSNZ.