Obfuscated Instructions for Software Protection

Akito Monden, Antoine Monsifrot, Clark Thomborson

November 2003
Abstract

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 update. We construct a computer system $M$ with the following properties: 1) the end-user may install program $P$ in any conveniently accessible area of $M$; 2) the program $P$ contains obfuscated 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 obfuscated instructions in $P$; thus, attempts to statically analyze the relation between instructions and their semantics will not succeed. We present a systematic method to construct 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.

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 [Chow, Johnson and Oorschot 2002]. 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 [Patrizio 1999]. 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 [The U.K. Parliament 2002]. 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 [Cohen 1993, Collberg and Thomborson 2002, Kanzaki et al. 2003]. 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.
This paper also extends the naïve key-search attacks. This paper is more resilient to brute-force enumerative attacks. That is, the proposal in this paper is more resilient to brute-force enumerative attacks.

In order to realize an obfuscated interpretation in $W$, we employ a FSM that takes as input an instruction stream $I$ with its 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 describe enhancements to our recently-proposed framework for constructing an interpreter $W$, which carries out obfuscated interpretations for a given program $P$ [Monden et al, 2003]. 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 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 $W$ 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 our original proposal [Monden et al, 2003], we had required the obfuscating opcode translation 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 obfuscating code by interpretation.

In some sense, the mechanism of our 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, Stinson 1995]; however, conventional stream ciphers can not simply be 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 instruction stream of $P_0$ as it executes on 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 should 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 described which is less restrictive than our original proposal. 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 $W$.
P₀: is a target program intended to be hidden from hostile users. For simplicity, we assume P₀ is written in a low level programming language, such as bytecode or machine code, where each statement in P₀ consists of a single opcode and (occasionally) some operands.

W₀: is a common (conventional) interpreter for P₀, such as a Java Virtual Machine, a Common Language Runtime or an x86 processor.

Pₓ: is a program containing obfuscated instructions whose semantics are determined during execution according to their context. This Pₓ is an equivalently translated version of P₀, i.e. Pₓ has the same functionality as P₀.

I: is an input of P₀ and Pₓ. Note that P₀ and Pₓ 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 Wₓ and a program translator Tₓ.

Wₓ: is a FSM-based interpreter that can evaluate obfuscated instructions of Pₓ according to the current state of the FSM built inside. This Wₓ is an extension of W₀ with a FSM unit of given specifications x.

Tₓ: is a program translator that automatically translates P₀ into Pₓ with respect to the specifications x.

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

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

Building an efficient Tₓ in a systematic manner is a fundamental part of this framework. Since Pₓ is quite different from ordinary programs, even though the program developer owns x, writing Pₓ from scratch is extremely difficult for the developer. In our framework, we provide a systematic method Tₓ to construct Pₓ from any given P₀ 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 P₀. This secret portion s is embedded in an interpreter Wₛ which is implemented in secure hardware, and attached to computer system Mₛ. The remaining part of the program (denoted Pₛ) is delivered to the user in
the usual way. This program is executed normally on
the CPU in $M_ε$, except for the secret portion which is
executed by making calls to the interpreter $W_r$. For
example, some of the arithmetic operations in $P_r$ may
be executed by $W_r$, possibly updating one or more
state variables held in $W_r$. Since the return value from
the calls to the interpreter $W_r$ may be used to control
branches and case statements in $P_ε$, much of the
control structure of $P_ε$ can be obscured. One
difficulty with this approach is that it does not allow
multiprogramming: while $W_r$ is holding state for $P_ε$,
no other program can be run on $W_r$. Another problem
is that any adversary who examines $P_ε$ will soon
discover how to call $W_r$. The adversary can then write
a program which makes similar calls to $W_r$ in various
orders. An analysis of the variability in the output of
$W_r$, when it is exercised in this systematic way, is
likely to reveal secrets of $W_r$. A final problem is that
updates to $P_ε$ 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, $W_r$ is built independent of
$P_0$; thus, many different programs run on a single
interpreter $W_r$, and any of the programs can be easily
updated without sending secret messages.

The most commonly-proposed method for hiding
interpretation is program encryption [Albert and
Morse 1984, Herzberg and Pinter 1987]. 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,$E(P_0)$ 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 $W_r$ in a non-accessible area of $M_k$’s hardware
is easily realized by adding a small FSM unit to
current hardware-based virtual machines [such as
picoJava, TinyJ, and Xpresso], modern
implementations of the x86 instruction set (which
translate it into a simpler microcode before
execution), and reconfigurable processors. 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 five types of design choices for the
FSM-based interpreter, which are dependent upon the
instruction set used for $P$. Let $Ins_{P0}$ and $Ins_{P_ε}$ be
the instruction sets for $P_0$ and $P_ε$, and let $L_{P0}$ and $L_{P_ε}$ be
the programming language for $P_0$ and $P_ε$, respectively.
We define five types of designs. Note: in our original
proposal we had not defined “Type 2.5”.

(Type 1) $Ins_{P_ε}$ is the same as $Ins_{P0}$ and all $P_ε$ have
correct static semantics in $L_{P0}$ (e.g. $P_ε$ would pass
Java’s bytecode verifier if $P_0$ were valid Java
bytecode) although the dynamic semantics are
determined during execution. Thus $P_ε$ is executable in
the original interpreter $W_0$ although its outputs would
be incorrect.

(Type 2) $L_{P_ε}$ has the same syntax as $L_{P0}$, but the static
semantics of $P_ε$ may be incorrect (e.g. if $L_{P0}$ is Java
bytecode, the stack signature of some opcodes in $P_ε$
may be incorrect). The number of different FSMs
that could be used to interpret $P_ε$ is larger than in
Type 1.

(Type 2.5) $L_{P_ε}$ has different operand syntax to $L_{P0}$;
individual opcodes in $P_0$ are translated into opcodes
in $P_ε$ with the same number of bytes; and the opcode
sets and encodings in $Ins_{P0}$ and $Ins_{P_ε}$ are identical.
Because the type and number of operands (and their
specifiers) associated with each opcode may differ from
$L_{P0}$, $P_ε$ is generally not a valid program in $L_{P0}$.
The number of different FSMs that could be used to
interpret $P_ε$ is larger than in Type 2.

(Type 3) $Ins_{P_ε}$ includes $Ins_{P0}$ 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) $Ins_{P_ε}$ differs completely from $Ins_{P0}$, however
there exists some (secret) many-to-one mapping
which transforms $Ins_{P0}$ into a Type-3 instruction set.
That is, $P_ε$ 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 2.5
designs.
2.2.2 Architecture

Figure 5 shows a suitable architecture for FSM-based interpreter, characterized by pipelined stages of interpretation. In this paper we focus on 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” obfuscated opcode into an unobfuscated opcode, passing it to a conventional opcode decode unit. Then, the translated opcode is decoded, and the number of operands to be fetched is determined. After 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 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 a by a special Type-3 instruction.

2.2.3 FSM unit

The FSM unit (denoted as $w_3$) is a DFA (Deterministic Finite Automaton) defined by 6-tuple $(Q, \Sigma, \Psi, \Delta, \Lambda, q_0)$ where

- $Q = \{q_0, q_1, \ldots, q_n\}$ is the states in the FSM.
- $\Sigma = \{c_0, c_1, \ldots, c_n\}$ is the input alphabet.
- $\Psi = \{\psi_0, \psi_1, \ldots, \psi_m\}$ is the output alphabet (interpretations for inputs).
- $\delta_i : \Sigma \rightarrow Q$ is the next-state function for state $q_i$.
- $\Delta = (\delta_0, \delta_1, \ldots, \delta_n)$ is the n-tuple of all next-state functions.
- $\lambda_i : \Sigma \rightarrow \Psi$ is the output function for state $q_i$.
- $\Lambda = (\lambda_0, \lambda_1, \ldots, \lambda_n)$ 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 $P_i$ is the same as that for $P_0$, so $Ins_{P_i} = Ins_{P_0}$. We assume $Ins_{P_i} = \sum O$ where elements $c_i \in \Sigma$ are obfuscated instructions, and $o_i \in O$ are non-obfuscated instructions. This means, $P_i$ 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.

In our Type-2.5 design, each underlined symbol $\psi_j$ 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 $c_{0b}, c_{1b}, \ldots, c_{b-1}$ are in the first class $C_1$ (of branching opcodes including non-conditional jump) and the remaining symbols $c_{b0}, c_{b1}, \ldots, c_{b1}$ are in the second class $C_2$ (of non-branching opcodes).

The 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 $j$, the length of the translated opcode $\lambda_i(c_j)$ is the same as the obfuscated opcode $c_j$ so that the opcode fetch unit can correctly fetch obfuscated opcodes. For example, in the instruction set of Intel x86 CPU family, the length of MOV opcode is the same as SUB but it differs from MOVZX [Intel 1999]. Thus the MOV opcode may be obfuscated as “sub” but not as “movzx”.
4. For all pairs of states $q_i, q_k$ there exists a “dummy instruction sequence” $d_{q_i q_k}$ with the following three properties. First, $d_{q_i}$ is a short sequence of (translated) instructions containing exactly one obfuscated instruction. Second, an FSM initially in state $q_i$ will be in state $q_k$ after it produces $d_{q_i}$ as output. Third, $d_{q_k}$ has no effective functionality.

![Figure 5. Pipelined stages of FSM-based interpreter](image-url)
Thus \( d_i \) is an efficiently executed no-op that forces the FSM to make any desired transition. Note that for any pair of states \( q_i, q_j \) there exists \( c_j \) such that \( \delta_i(c_j) = q_j \), because the next-state function \( \delta \) is a bijection. The obfuscated instruction in \( d_i \) is \( \lambda_i(c_j) \).

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

Figure 6 shows a simple example of \( w_x \) where

\[
\begin{align*}
Q &= \{q_0, q_1\} \\
\Sigma &= \{\text{add}, \text{sub}\} \\
\Psi &= \{\text{add}, \text{sub}\} \\
\Delta : \delta_0(\text{add}) &= q_1, \delta_0(\text{sub}) = q_0, \delta_1(\text{add}) = q_0, \delta_1(\text{sub}) = q_1 \\
A : \lambda_0(\text{add}) &= \text{sub}, \lambda_0(\text{sub}) = \text{add}, \lambda_1(\text{add}) = \text{add}, \lambda_1(\text{sub}) = \text{sub}
\end{align*}
\]

This \( w_x \) takes an opcode \( c_j \in \{\text{add}, \text{sub}\} \) as an input, translates it into its semantics \( c_j \in \{\text{add}, \text{sub}\} \), and outputs \( \Psi \). 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 \text{add} or \text{sub} according to its context.

**2.2.4 Program translator**

In order to utilize the FSM-based interpreter \( W_x \), a program translator \( T_x : P_0 \rightarrow P \) 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 \( p := 1+2+3+\ldots+n \). 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 \( P \) 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 \( P \). Our \( w_x^{-1} \) is the DFA defined by 6-tuples \( (Q', \Sigma', \Psi', \Delta', A', q_0) \) where

\[
\begin{align*}
Q' &= Q = \{q_0, q_1, \ldots, q_{n+1}\} \text{ is the states in the FSM.} \\
\Sigma' &= \Psi' = \{c_0, c_1, \ldots, c_{n+1}\} \text{ is the input alphabet.} \\
\Psi' &= \Sigma = \{c_0, c_1, \ldots, c_{n+1}\} \text{ is the output alphabet.} \\
\Delta : \delta_i(\Sigma') &\rightarrow Q' \text{ is the next-state function for state } q_i, \text{ where } \delta_i(\Sigma_j) \text{ has the value } \delta_i(\lambda_i^{-1}(c_j)) \text{ for all } i, j.
\end{align*}
\]
Δ' = (δ'k, δ'j, ..., δ'n+1) is the n-tuple of all next-state functions.

λ'j: Σ' → Ψ' is the output function for state qj, where each λ'j: Σ' → Ψ' has the value λ'j(qj) for all i, j.

A' = (λ'0, λ'1, ..., λ'n+1) is the n-tuple of all output functions.

q0 ∈ Q' is the starting state of the FSM.

Figure 9 shows an example of wx⁻¹ corresponding to wx of Figure 6. As shown in Figure 9, wx⁻¹ has the same number of states and transitions as wx.

Next, we give a procedure for the translation T*: P₀ → Pₙ. Figure 10 shows this procedure where:

PC is a program counter (we assume PC is a line number of P₀).

c odeP₀(PC) is an instruction in P₀ at PC.

c odePₙ(PC) is an instruction in Pₙ at PC.

qₙ ∈ Q is a state of wx⁻¹.

state(PC) is a state in which codePₙ(PC) was interpreted.

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

Figure 11 shows an example of Pₙ, translated from P₀ of Figure 8. In this example, a dummy instruction “add p, 0” is inserted into Pₙ to force the state transition q₁ → q₀ so that w comes to q₀ every time the execution reaches the entry point of loop.

3 Case Study

3.1 Program translation

In this section we explain a more complex example, in which we execute the procedure T*: P₀ → Pₙ of Figure 10 using the inverse interpreter wx⁻¹ given in Table 1. This wx⁻¹ is designed for programs written in an Intel x86 instruction set. We use the AT&T syntax (GNU assembler format) to write assembly code in L₀ and Lₙ. Our sample wx⁻¹ has eight states Q' = {q₀, q₁, ..., q₇} with q₀ a starting state, and has eight types of instructions Σ': {imp, jne, pushl, decl, movl89, subl, movl89, leal}. Here, “movl89” indicates MOV instructions whose opcode byte are “89”, and “movl89” indicates “8B” opcode as well. Two instructions (imp and jne) are branching instructions. The other six are non-branching instructions. For each instruction in Table 2, a binary

Let state(k) := NULL for all k of P₀

Let q₀ := q₀

Set PC to the entry point of P₀

loop:

If PC = exit of P₀ then goto resume

If state(PC) ≠ NULL & & state(PC) ≠ qₙ then {

Call choose&insert_dummy

Goto resume

}

Let state(PC) := qₙ

If codeP₀(PC) ∈ Σ' then { /* obfuscated instruction */

Interpret codeP₀(PC) via wx⁻¹, i.e.

Let q₀ := δ'(codeP₀(PC))

Let codePₙ(PC) := λ⁻¹(codePₙ(PC))

} else { /* non-obfuscated instruction */

Let codePₙ(PC) := codePₙ(PC)

}

If codePₙ(PC) = branching instruction then {

If codePₙ(PC) ≠ non-conditional jump then {

Do push(PCfalse) where PCfalse is a line number of next instruction in false branch

Let state(PCfalse) := qₙ

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()

q₀ := state(PC)

Goto loop

choose&insert_dummy:

Let PCprev := previous value of PC

If codePₙ(PCMprev) = non-branching instruction then {

Choose q₂ ∈ Σ' that satisfies δ'(q₂) = state(PC)

Let d₂ := a sequence of dummy instructions for q₂

Let dₙ := λ⁻¹(d₂)

Insert d₂ into Pₙ right after the line number = PCprev

} else {

Choose k that satisfies δ'(codePₙ(PCprev)) = state(PC)

Choose q₂ ∈ Σ' that satisfies δtrue(PCprev)(q₂) = qₙ

Let d₂ := a sequence of dummy instructions for q₂

Let dₙ := λ⁻¹(d₂)

Insert d₂ into Pₙ at the line number = PCprev

state(PCprev) := qₙ

}

return

Figure 10. Procedure for T*: P₀ → Pₙ

(hexadecimal) representation of the opcode is shown. Please see [Intel 1999] for detailed information on instruction semantics.
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 14, 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 \( L_{P_0} \) into instructions, since the correct number of operands required for each opcode \( L_{P_0} \) differs from that in \( L_{P_0} \). (Indeed, \( L_{P_0} \) might not even have a consistent syntax.)

2. Instructions in \( P_x \) do not have static binding to their semantics. For example, “pushl” in line 2 is interpreted as “movl" via \( w_s \) (see the same line in Figure 13), but in line 24, it is interpreted as “lea\(". 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 “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. (In addition, if a translated program \( P_x \) contain a dummy instruction sequence for a branching instruction, then the apparent control flow of \( P_x \) is more complex than \( P_0 \).)

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 (jne) opcode with a GOTO (jmp) 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 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.

We do not expect to be able to prevent expert and well-resourced adversaries from gaining level-1 understanding. However, as argued below, our protection scheme in conjunction with other obfuscations will prevent adversaries with considerable knowledge, resources and motivation from ever gaining level-3 understanding. Weaker adversaries will be unable to gain level-2 or even level-1 understanding, unless they are very persistent.

---

Figure 13. \( P_0 \) in AT&T syntax

```assembly
.globl sumodd
.type sumodd,@function
.example:
1 q0 pushl %ebp
2 q2 movl $0x800, %esp
3 q4 subl %ebp, -4(%esp)
4 q4 movl $1, -8(%ebp)
5 q4 .L3:
6 q4 movlSB -8(%ebp), %eax
7 q7 cmpl 8(%ebp), %eax
8 q7 jle .L6
9 q7 jmp .L4
10 q7 .L6:
11 q7 .L5:
12 q7 movlSB -8(%ebp), %edx
13 q7 movlSB %edx, %eax
14 q4 sarl $31, %eax
15 q4 shrl $31, %eax
16 q4 leal (%eax,%edx), %eax
17 q4 sarl %eax
18 q4 movlSB %eax
19 q4 subl %eax
20 q0 movlSB %eax
21 q1 cmpl $1, %eax
22 q1 jne .L5
23 q3 movlSB -8(%ebp), %edx
24 q4 leal -4(%ebp), %eax
25 q5 addl %edx, (%eax)
26 q5 .L5:
27 q3 leal -8(%ebp), %eax
28 q0 incl (%eax)
29 q0 jmp .L3
30 q0 .L4:
31 q0 movlSB -4(%ebp), %eax
32 q3 leave
33 q3 ret
```

Figure 14. \( P_x \) in AT&T syntax

```assembly
.globl sumodd
.type sumodd,@function
.example:
1 decl %ebp
2 pushl %esp, %ebp
3 subl $8, %esp
4 movl $0, -4(%ebp)
5 movl $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 %eax
15 shrl %eax
16 leal (%eax,%edx), %eax
17 sarl %eax
18 sall %eax
19 movlSB %eax
20 movlSB %edx
21 cmpl $1, %eax
22 jmp .L5
23 leal -8(%ebp), %edx
24 pushl -4(%ebp), %eax
25 addl %edx, (%eax)
26 .L5:
27 movlSB -8(%ebp), %eax
28 incl (%eax)
29 jne .L3
30 .L4:
31 leal -4(%ebp), %eax
32 leave
33 ret
```
We characterize an adversary’s knowledge and resources along several dimensions (labeled A, B, etc.), 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 naïve 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. Our level-3 adversary is expert and extremely well-resourced, and our level-4 adversary is in possession of powerful custom software.

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 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. The adversary is able to reverse-engineer a software implementation of $W_x$, so that it is possible to collect output traces from $W_x$, and to inject arbitrary input for translation by $W_x$.

4. The adversary has source code for a generic interpreter $W(x)$ which emulates $W_x$ for any $x$, and a generic translator $T(x)$ which implements $T_x$ for any $x$.

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 has a generic interpreter $W(x)$, and knows how to use this to record a complete trace of the output of $W_x$ for any $x$. The adversary also has a generic translator $T(x)$ whose outputs can be recorded.

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” $d_j$ 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-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) in our Type 2.5 FSM design, although the adversary will certainly make some mistakes in recognition of branching opcodes wherever instruction boundaries are obscure in the obfuscated code due to the differing operand syntax in $L_{PK}$ and $L_{PN}$. For example a 0xEB byte in the obfuscated sequence is reasonably likely to be an obfuscated branch opcode but it may also be an operand.

The adversary might examine $O(n^2)$ loops to be reasonably certain of having discovered all suffixes, so hypothesizing $d_j$ may take days but not months if $n = 100$. 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 cryptographically-skilled level-1 attacker could then build up a (hypothesized) list $d_{P_{ij}}$ of obfuscated dummy sequences by substituting all (hypothesised) $c_i$ for the $c_i$ in each (hypothesized) sequence $d_j$. 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 $d_{P_{ij}}$, $d_{L_{ij}}$ 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 Type 2.5 FSM $W_x$. We have not done a complete cryptographic analysis, however our preliminary analysis indicates that $O(n^2)$ observations and controls would suffice for an attack of the type described above, on an $n$-state Type 2.5 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, for example by using multiple “dummy sequences” for each instruction, by randomizing 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 $P_{00}$), by using a Type-2 or Type-2.5 design without a partition between branching and non-branching opcodes, by using a Type-3 FSM to make it harder for the attacker to recognize no-op suffixes and prefixes, or by using a Type-4 FSM to increase $n$. We intend to explore these options in future work.

The “crack” described above for a level-1 attacker gives them level-2 understanding of a single machine $M_x$, for they can predict how a single FSM $W_x$ will translate arbitrary inputs – including the obfuscated program $P_x$! (Note: the attacker must 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 $\lambda(c_i)$ of an instruction $c_i$ 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. 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 2.5” translation of an x86 assembly-language $P_0$ into an “x86-like” $P_x$ 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 interpreters of Type 1, 3 and 4, and we intend to clarify their advantages and shortcomings.

6 References


Table 1. Example of FSM \( w_{v^{-1}} \)

<table>
<thead>
<tr>
<th>State</th>
<th>Input / Output</th>
<th>transition</th>
<th>State</th>
<th>Input / Output</th>
<th>transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>EB jmp / 75 jne</td>
<td>( q_4 )</td>
<td>EB jmp / 75 jne</td>
<td>( q_6 )</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>EB jmp</td>
<td>( q_4 )</td>
<td>55</td>
<td>pushl %ebp / 8D decl %ebp</td>
<td>( q_5 )</td>
</tr>
<tr>
<td>55</td>
<td>pushl %ebp / 4D decl %ebp</td>
<td>( q_5 )</td>
<td>4D</td>
<td>decl %ebp / 55 pushl %ebp</td>
<td>( q_5 )</td>
</tr>
<tr>
<td>4D</td>
<td>decl %ebp / 55 pushl %ebp</td>
<td>( q_5 )</td>
<td>89</td>
<td>movl89 / 89 movl89</td>
<td>( q_5 )</td>
</tr>
<tr>
<td>29</td>
<td>subl / 29 subl</td>
<td>( q_6 )</td>
<td>29</td>
<td>subl / 8B movl89</td>
<td>( q_6 )</td>
</tr>
<tr>
<td>8B movl89 / 8D leal</td>
<td>( q_7 )</td>
<td>8B movl89 / 8D leal</td>
<td>( q_7 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8D leal / 8B movl89</td>
<td>( q_7 )</td>
<td>8D leal / 8B movl89</td>
<td>( q_7 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. List of sequence of dummy instructions

<table>
<thead>
<tr>
<th>Sequence of dummy instructions</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp .Lx any instructions</td>
<td>0</td>
</tr>
<tr>
<td>.Lx</td>
<td></td>
</tr>
<tr>
<td>ine .Lx</td>
<td>1</td>
</tr>
<tr>
<td>addl $0,%eax</td>
<td></td>
</tr>
<tr>
<td>.Lx</td>
<td></td>
</tr>
<tr>
<td>pushl %eax</td>
<td>2</td>
</tr>
<tr>
<td>pop %eax</td>
<td></td>
</tr>
<tr>
<td>.Lx</td>
<td></td>
</tr>
<tr>
<td>decl %eax</td>
<td>3</td>
</tr>
<tr>
<td>incl %eax</td>
<td></td>
</tr>
<tr>
<td>.Lx</td>
<td></td>
</tr>
<tr>
<td>movl89 $0,%eax</td>
<td>4</td>
</tr>
<tr>
<td>movl89 %eax,%ebx</td>
<td></td>
</tr>
<tr>
<td>subl %ebx,%eax</td>
<td>5</td>
</tr>
<tr>
<td>.Lx</td>
<td></td>
</tr>
<tr>
<td>movl89 !-4(ebp),%eax</td>
<td>6</td>
</tr>
<tr>
<td>leal !-4(ebp),%eax</td>
<td>7</td>
</tr>
</tbody>
</table>

* these registers must be dead