Control Flow Based Obfuscation

Jun Ge
Dept. of Computer Science
Iowa State University
Ames, IA 50011
junge@iastate.edu

Soma Chaudhuri
Dept. of Computer Science
Iowa State University
Ames, IA 50011
chaudhur@iastate.edu

Akhilesh Tyagi
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
tyagi@iastate.edu

ABSTRACT

A software obfuscator is a program $\mathcal{O}$ to transform a source program P for protection against malicious reverse engineering. $\mathcal{O}$ should be *correct* ($\mathcal{O}(P)$ has same functionality with P), *resilient* ($\mathcal{O}(P)$ is resilient against attacks), and *effective* ($\mathcal{O}(P)$ is not too much slower than P). In this paper we describe the design of an obfuscator which consists of two parts. The first part extracts the control flow information from the program and saves it in another process named *Monitor-process*. The second part protects Monitor-process converting it into an Aucsmith like self-modifying version. We prove the correctness of the obfuscation scheme. We assess its resilience and efficiency to show that both are high. This supports the claim that our approach is practical.

Categories and Subject Descriptors: D.2 [Software Engineering]: Miscellaneous

General Terms: Security, Design, Experimentation.

Keywords: software obfuscation, control flow.

1. INTRODUCTION

In 2005, the IDC and Business Software Alliance(BSA)'s annual study on software piracy [2] shows that, although the world piracy rate decreased slightly to 35%, $31 billion was lost due to piracy globally. Computer scientists and software companies have invested significant money and energy to protect intellectual property (IP) embedded in software.

Collberg & Thomborson [8] provide an overview of software protection techniques. The three main technologies for software protection are *watermarking*, *obfuscation*, and *tamper-resistance*. *Software Obfuscation* generally refers to the process of transforming a program P through an obfuscator $\mathcal{O}$ into $\mathcal{O}(P)$ such that an adversary can derive no more information from a white-box observation of $\mathcal{O}(P)$ than from a black-box observation of P. Of course, P and $\mathcal{O}(P)$ must have equivalent functionality. A robust widely-accepted definition of obfuscation does not exist. The ambiguity in the preceding description of obfuscation has to do with the quantification of "information", "white-box", and "black-box". Barak *et al.* [3] proved that an obfuscator does not exist if the obfuscated program needs to serve as a "virtual black box", which means that the obfuscated program reveals nothing more than its functionality. Although this result may imply the nonexistence of an obfuscater, the "virtual black box" requirement of an obfuscator is very restrictive. With more relaxed constraints, obfuscators do exist, and several researchers have achieved positive results.

There are three broad categories of obfuscation methods. *Lexical Obfuscation* typically tries to scramble the identifiers and is used in software like KlassMaster and JZipper for java program protection. *Data Obfuscation* modifies data structures to gain security [7]. *Control Flow Obfuscation* hides the control flow information of the program. Cloakware^{TM} [4] is an example of the control flow obfuscation category through introduction of a dynamic dispatcher. Collberg *et al.* [6] proposed another way to obfuscate the control flow of an application by inserting redundant conditionals and loops.

Diversity which is important for a species to survive is also used in software obfuscation. Forrest *et al.* proposed diversity technology as a tool to develop more robust and secure program [10]. Diversity is also used by Cloakware^{TM} [4]. Aucsmith [1] proposed another powerful obfuscation and anti-tamper mechanism by combining Integrity Verification Kernels and Interlocking Trust Mechanism.

In this paper, we develop a control flow obfuscation paradigm based on a two-process model. The control flow information (CFI) is stripped out of the obfuscated program through a permutation of static blocks of program/text. Another concurrent monitor process is created which contains the static permutation. The obfuscated program (program process or P-process) queries the monitor process (M-process) for the correct address at each obfuscated juncture. The P-process and the M-process communicate with each other through inter-process communication (IPC) mechanism supported in Linux. Since the monitor process is a much smaller program than the program process, more expensive and elaborate protection mechanisms can be applied to it. We use a self-modifying version of monitor process, which is a simplified version of Aucsmith's scheme [1]. This obfuscator was implemented in gcc. We have tested it on a variety of programs to assess the performance overhead.

The self modifying image of the M-process protects it from **dynamic** reverse engineering. Dynamic reverse engineering is a widely-used attack method. An adversary runs the obfuscated program repeatedly to observe the program's image in the memory, to trace and analyze its control flow and data flow, and reverse-engineer the program image into a high level language. Without hardware support, the dynamic attacks seem to be hard to prevent. Static analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DRM'05, November 7, 2005, Alexandria, Virginia, USA.
Copyright 2005 ACM 1-59593-230-5/05/0011 ...$5.00.

based reverse engineering of our schema can be shown to be PSPACE-complete along the lines of [25]. The reachability problem within the M-process can be reduced to the acceptance problem for a linear bounded deterministic Turing machine (LBDTM) which is PSPACE-complete. The self-modifying code schema of M-process provides resistance to dynamic attacks. The main idea behind the self-modifying schema is to always hide most of the source program's code by modifying it continuously. M-process is compiled in such a way that it is divided into several cells (groups of basic blocks) of the same size. Each cell is obfuscated with one or several keys. When M-process is initially loaded into the memory, only the entry cell is open in plain text. All the other cells are encrypted. During the execution, every time the control flow goes from cell C_a to another cell C_b, the M-process (the text section of its image in the memory) modifies itself, encrypts C_a, decrypts C_b, and continues executing the instructions in C_b. During the self modification, cells other than C_a and C_b are also modified but still kept encrypted.

The rest of this paper is organized as follows. Section 2 provides the background and related work in software obfuscation. Section 3 describes the CFI-hiding scheme. Section 4 discusses the self-modifying scheme for M-process. Section 5 evaluates the efficiency of the obfuscator implementation. Finally, Section 6 summarizes and concludes the paper.

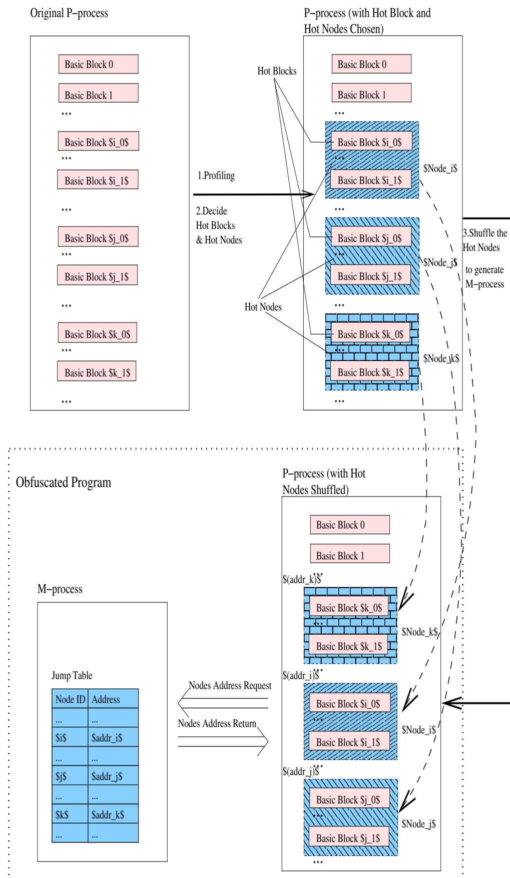

Figure 1: CFI Extraction & Embedding into Monitor-Process

2. BACKGROUND & RELATED WORK

Three main kinds of malicious host attacks on software embedded IP are: software piracy, malicious reverse engineering, and software tampering. Software obfuscation prevents the second kind of attack, malicious reverse engineering. Software obfuscation is a kind of code transformation [19]. It transforms an executable into a new one which has the same functionality but is hard to reverse-engineer.

An obfuscator should have three important properties. **(1)** The most important requirement is that it should retain the functionality of the program. **(2)** The obfuscated program should not be too much slower or use too much more space than the original one. **(3)** It should take the attackers more time to reverse-engineer the application than developing a new one by themselves. We give a definition of software obfuscator, adapted from [5] and [3].

An algorithm $\mathcal{O}$ is a ***Software Obfuscator*** *if the following three conditions hold:* **(i)** *For any given* ***source program*** *P, $\mathcal{O}$ transforms P into another* ***protected program*** *P' such that* **(a)** *if P fails to terminate with an error condition, then P' may or may not terminate.* **(b)** *Otherwise, P' must terminate and produce the same output as P. If the functionality is maintained, we say that P is* ***correct***. **(ii)** *The run time of P' should be at most polynomially larger than that of P.* **(iii)** *The time taken by an attacker to recover P from P' should be at least as large as the time to develop P from scratch.*

As we had stated earlier, the three general styles of obfuscation are lexical, data and control-flow obfuscation. Although the lexical obfuscation and data obfuscation frustrate attackers to some extent, control-flow obfuscation is by far the most resilient obfuscation method, because most attack efforts are spent on the control flow analysis. Control flow obfuscation methods falls into two main categories. The first category introduces spurious code blocks into the source program to obscure the real CFI. Collberg *et al.* [6] implemented such an obfuscator. In their work opaque constructs are manufactured to embed intractable static analysis problems into the obfuscated program. The second category extracts CFI from the code and hides it somewhere else. In Stanley Chow *et al.*'s work [4], the obfuscator divides the source program into several pieces, adds new dummy chunks into it, disperses these chunks, and generates a dispatcher aware of the true control flow between all the code chunks. They proved that the CFI-hiding dispatcher is resilient to static analysis based reverse-engineering attacks.

To evaluate an obfuscator, we focus on its *functionality*, *efficiency* and *resilience*. It goes without saying that an obfuscator must maintain the original *functionality*. In a CFI obfuscation scheme, we could prove that the obfuscation scheme is correct by showing that the CFI is preserved during the process of obfuscation.

For *efficiency*, we use simple definitions of *cost* (adapted from [5]) to account for the extra execution time and space of $\mathcal{O}(P)$ compared to P.

Definition 1 (Obfuscator Cost). *Given an obfuscator $\mathcal{O}$ and a program P, the* ***Time Cost*** *$C_t(\mathcal{O}, P)$ and the* ***Space Cost*** *$C_s(\mathcal{O}, P)$ are defined as*

$$C_t(\mathcal{O}, P) = \frac{T(\mathcal{O}(P))}{T(P)} - 1 \quad (1)$$

and

$$C_s(\mathcal{O}, P) = \frac{S(\mathcal{O}(P))}{S(P)} - 1 \quad (2)$$

where $T(M)$ is the execution time of program M and $S(M)$ is the space taken by M.

One way to establish an obfuscator's *resilience* against static attacks is to show that a known hard problem would have to be solved in order for reverse-engineering to succeed. For example, Zakharov [25] reduced a PSPACE-Complete problem to the reachability problem of cloak program dispatcher, which hides the control flow information of the source program. A similar method is used in [21]. In his Ph.D thesis, Wang developed a *one-way transformed* compiler which hides the static control-flow graph of the source program and showed that analyzing the the transformed program is NP-hard. We adapt Zakharov's reduction in Section 3.6 to show that reverse-engineering of M-process is PSPACE-complete.

3. CFI EXTRACTION FROM P-PROCESS TO M-PROCESS

Our approach towards software obfuscation focuses on control flow obfuscation and consists of two parts. The first part hides the CFI of the source program by compiling it into two co-processes: a main program (P-)process and a monitor (M-)process. The two processes *together* are the obfuscated program. The P-process implements the main functionality and it is similar to the source program. A part of P-process' CFI is extracted and stored in a table named Jump Table. This jump table is embedded into the M-process. P-process communicates with M-process requesting a jump table lookup whenever it reaches a point with missing CFI.

Section 4 will discuss the second part of our obfuscator, where a self-modifying scheme is deployed to ensure that only a small part of the code is observable to an adversary.

3.1 Basic Structure

The basic structure of our CFI hiding scheme is illustrated in Figure 1. Several definitions are given first.

Definition 2 (Hot Node). *A Hot Node is a group of one or more adjacent basic blocks that are kept together in their original static form during the obfuscation. The entire hot node is shuffled as a unit. The* size *of a hot node is the number of basic blocks in the hot node.*

Definition 3 (Hot Block). *A basic block B in P-process is called a* Hot Block *if B is executed with high probability (a user supplied threshold) during the execution of P-process. Each hot block forms a seed of a hot node as its start block.*

There are four steps in the CFI extraction as shown in Figure 1. We first profile the source program to determine basic block instantiation probabilities to derive hot blocks and hot nodes. With the help of Machine-SUIF ([18], [12], [24]) infrastructure for constructing compiler back ends, we developed passes to derive profiling information. We also ensure that the generated assembly code contains only labeled addresses instead of relative or absolute addresses. This allows for relocatability for the later compiler passes.

The second step is to extract hot blocks and hot nodes. Based on the profiling information, we select a set of hot blocks. The hot nodes automatically follow from the selection of hot blocks. Given the efficiency implications of hot blocks (each hot node in P-process initiates one communication with the M-process), the user can specify the maximum number of hot blocks n the obfuscator can use.

The third step is to shuffle/permute the hot nodes so as to hide the true static layout of the source program. The shuffling is performed at assembly level code. Hot nodes are shuffled randomly. The correspondence between the original ordering of the hot nodes to the shuffled ordering constitutes the hidden CFI. This is recorded in the jump table. New instructions are inserted at the hot node boundaries to request CFI from the M-process.

M-process is initialized right after P-process is started. It then enters a finite wait-loop waiting for P-process' address (CFI) requests. M-process has exclusive access to the jump table. The M-process exits just before the P-process terminates.

3.2 Implementation Steps

In our implementation, we developed several compiler passes to fulfill the task of program obfuscation. Most passes are generated in the following way. At first, the CFG Intermediate Representation (IR) of the source program is generated with the help of Machine-SUIF. Then annotations are added at proper locations of the CFG IR such as the beginning and the end of each basic block. Machine-SUIF then generates assembly code with the annotation. Generally each annotation is identified with the unique ID of the basic block to which it is attached. One thing to note here is that the assembly code is generated so that all the targets of Control Transfer Instructions (CTI, such as `jmp`, `jc`) are labeled addresses. Keeping the hot blocks and hot nodes relocatable through labeling simplifies the shuffling process.

Several other passes were developed in Perl to manipulate the assembly code generated by Machine-SUIF passes. All the code insertion, modification and shuffling are completed in Perl except for the annotation. Perl was chosen because of its powerful text processing and regular expression support.

We also developed several other Perl scripts to process the data generated during obfuscation. For example, the program profiling generates profiling information in a format not suitable for later passes. A Perl script was written to refine the data into a more manageable format.

3.3 Hot Blocks and Hot Nodes Extraction

The n most frequently instantiated blocks are chosen as hot blocks, where n is the maximum number of hot nodes specified by the user. Recall that these hot blocks form the start blocks in the hot nodes.

The choice of a hot block is important for both obfuscation and performance. Our goal is to maximize both resilience and efficiency. First, we need to choose an appropriate number of hot nodes. A higher number of hot nodes leads to a higher level of obfuscation, but at the cost of higher overhead. This is why we let the user be the decision maker. Let *shuffling rate* denote the ratio of the number of hot nodes to the number of total blocks n/N for a program with N basic blocks.

There are some additional constraints on the size and position of a hot node. First, a node should not straddle the border of functions. That is, all the blocks of a node should belong to the same function. Second, a node should not contain more than one hot block. Otherwise the control relation between the two hot blocks is not hidden well.

Once a hot block b is chosen, it corresponds to a hot node of certain size starting at b. The hot node starting with b has size $s(b)$ given by:

$$s(b) = \min(d, \max(1, d(b) * n/r(b))) \tag{3}$$

Here $r(b)$ is the node b's rank in the profiling data, $d(b)$ is

the maximum possible size of the node starting at b. This is bounded by the location of the next hot block, the number of basic blocks between b and the next hot block. n is the number of selected hot nodes. Note that $s(b)$ is proportional to r, because smaller the rank, more frequently the node is accessed. Due to spatial locality in block accesses, it is very likely that the blocks in the vicinity of b are also accessed equally frequently. s is also proportional to d. This tries to make nodes as large as possible.

3.4 Shuffling/Permuting the Hot Nodes

The hot nodes are shuffled randomly. The extracted hot nodes are shuffled by the fisher-yates shuffling algorithm [17]. Once the shuffling permutation is finalized, the hot nodes are moved to reflect this permutation in the assembly code.

We generate another Machine-SUIF pass for annotations to help the shuffling of the nodes. This time each block is marked by two annotations, one at its beginning and one at its end. Each node N_i is marked with two annotation, stt_i and end_i. After the annotation, the assembly code is processed by a script named `shfl.pl`. Table 1 is the pseudo-code `shfl.pl` generated at the location where N_i is originally located. In the shuffling, N_i is substituted by N_j. Function `_getAddr` is implemented in another source file whose task is to contact M-process (MP) for address queries. All the examples in this paper use Intel X86 assembly.

Table 1: Pseudo-code After The Shuffling Where N_i Is Substitute by N_j

1. push flags and registers onto stack
2. push stt_i onto the stack, call function`_getAddr` and clean argument stt_i
3. `jmp %eax`
4. label `gl_`stt_j :
5. restore flags and registers
6. code of N_j
7. push end_j onto the stack, call function`_getAddr` and clean argument end_j
8. `jmp %eax`
9. label `gl_`end_i :
10. restore flags and registers
11. original code after N_i

Before calling `_getAddr`, flags and registers used by `_getAddr` are pushed onto the stack (step 1). Then `_getAddr` is called with stt_i as the argument (step 2). After returning from `_getAddr`, `%eax` holds the start address of N_i, so the `jmp` instruction in step 3 directs the control flow to the new address of N_i.

After step 3, code similar to step 4 - 8 (with j substituted by i) is executed at N_i's new location. Step 5 restores the flags and registers before running code of N_i. After N_i's code is finished (step 6), the control flow should return to the code that originally follows N_i. This is done in step 9 where `_getAddr` is called again with argument end_i. Then the flags and registers are restored at step 10. At last the original code after N_i is executed at step 11. This maintains equivalence of control flow before and after shuffling. The only difference is that the addresses of the hot nodes are hidden after shuffling.

3.5 M-process Generation

The main task of M-process is to handle address requests from P-process. Since the M-process contains the program secret in the form of the jump table, it needs the extra protection. We describe the self-modifying code scheme in Section 4. Here we only show M-process's interface, i.e., its creation and activation in P-process and its functionality.

Instructions are added into the P-process to create the M-process, and to send jump table lookup requests. The code modification is done by the same script that shuffled the code, `shfl.pl`. M-process is initialized (`fork`ed) at the entry point of P-process' `main()`. Recall that there is a special annotation there so that `shfl.pl` could insert an instruction `call _startMp` at the correct point. Function `_startMp` is implemented as a `C` program which creates the M-process.

Every time the M-process gets a block id, it returns the address of the hot node through a jump table lookup. At the P-process side, the boundaries of the hot nodes are the places where the CFI is hidden and instructions `call _getAddr` are inserted by `shfl.pl`, as described in step 2 and step 7 of Table 1. The function `_getAddr` sends address lookup requests to the M-process. The address returned by `_getAddr` (returned in turn by M-process) is stored in register `%eax` so that instruction `jmp %eax` can redirect the control flow to the new address of the hot node. (See step 3 and step 8 in Table 1). The insertion of `call _getAddr` is described in step 2 and step 7 in Table 1.

3.6 Resilience of the CFI-hiding Scheme

This section discusses the resilience the CFI-hiding scheme. We first prove that a specific reachability problem within our obfuscator required of any static analysis driven reverse engineering is PSPACE-Complete.

A linear bounded deterministic Turing machine (LBDTM) (see [11]) M is an conventional Deterministic Turing Machine (DTM) (see [14]) with the restriction that no computation uses more than $n+1$ tape cells. There are input boundary markers, left marker $\vdash$ and right marker $\dashv$, at the two ends of the input string. The head of the machine is not allowed to move left of $\vdash$ or right of $\dashv$. Note that a language L is PSPACE-complete if L is in PSPACE and for every language $L' \in PSPACE$, $L' \leq_p L$. The PSPACE-complete language which is reduced to our obfuscator reachability problem is Linear Bounded Automaton Acceptance (LBTM-ACC) [11]: LBTM-ACC $= \{< x, M >: x$ is accepted by M, where M is a $LBTM$ and x is an input to $M\}$. In [16], Karp proved that LBTM-ACC is in PSPACE-complete.

Reachability Problem for M-process: The essence of M-process behavior is to receive an address lookup request and to return the looked-up address to the P-process. Hence, it is a deterministic automaton with an output function. The input of this automaton is the ID of a hot node and the output is the address of the hot node. Because the total number of blocks in the obfuscated software is limited (finite), the M-process is a Deterministic Finite Automaton (DFA) A, which is a 6-tuple: $< S, \Lambda, \Psi, \varsigma, s_0, F, \tau >$, where S is a finite set of states, Λ is the input alphabet (hot node label space), Ψ is the output alphabet, ς is the transition

function, $\varsigma : S \times \Lambda \rightarrow S$, s_0 is the initial state, F is a set of final states and τ is the output function, $\tau : S \rightarrow \Psi$.

Given two states s and s', we say that s' is *reachable* from s if there exists a sequence of transitions of M-process DFA leading from state s to s'. Intuitively, any assertion dealing with the reachability of basic blocks within the obfuscated P-process ("is the basic block B' reachable from the basic block B?" is the kind of question at the heart of static reverse engineering) translates into the aforementioned reachability problem within the M-process.

THEOREM 3.1. *M-process reachability (MPROCESS-REACH) $\in$ PSPACE.*

PROOF OMITTED.

THEOREM 3.2. *LBDTM-ACC $\leq_p$ MPROCESS-REACH.*

The proof is very similar to the one used in [25], and we omit it for brevity.

4. SELF-MODIFYING MONITOR-PROCESS SCHEMA

Many previous research efforts in obfuscation assume that there is a secure component [26] which is hidden from observation. The secure component may reside in a smart card or in a secured server. In this section, we describe a self-modifying code schema, in which the M-process image in memory keeps modifying itself. This protection approach offers good obfuscation without any specialized hardware support.

4.1 Basic Structure

The binary executable of the obfuscated program (obfuscated M-process in this case) is divided into several pieces, each of which contains one or more basic blocks. Each piece of code is called a *cell*. During M-Process execution, the control flow sometimes exits from one cell and enters another. We call this a *control flow switch*. The source of control flow switch is called a *source block*. The block executed after the control flow switch is called a *target block*. The cell containing the source block is called a *source cell* and the one containing the target block is called a *target cell*. The cell containing the *entry block* of the M-process is called the *entry cell*. An exit block of the M-process is contained in an *exit cell*. Note that there is only one entry block and one entry cell, but there may be many exit blocks and exit cells.

There are keys embedded at the end of each source block. If there is only one branch target leading outside the source cell B, only one key is embedded in B. If there are two branch targets leading to cells outside B, B contains two keys. Each key corresponds to a control flow switch. All the keys have an important property that after XOR'ing a proper key with the *whole* M-process' image, the source cell is encrypted, the target cell is decrypted and all the others are changed but still remain encrypted.

When M-Process starts, one or more blocks in the entry cell C_0 are executed until the first control flow switch s occurs. Let B_{new} be the target block, C_{new} be the target cell of s, and the key corresponding to s be $k_{0,new}$. Before B_{new} is executed (in fact B_{new} can not be executed at this moment because it is encrypted), $k_{0,new}$ is used to XOR the whole image. The XOR operation will close C_0, open C_{new}, and change all the other cells. Then the control flow goes to B_{new} and executes one or more blocks until another control flow switch occurs. This process continues until an exit block is executed and the M-Process terminates. At each point in execution, there is always exactly one cell that is observable.

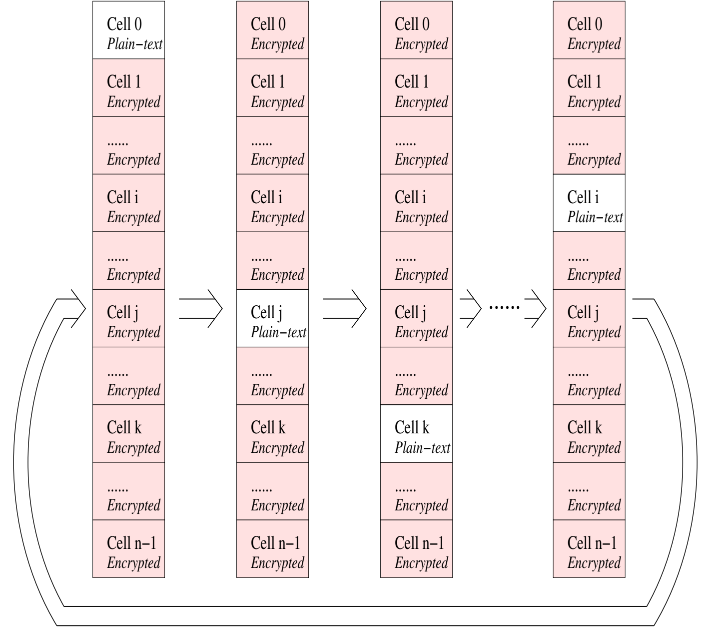

Figure 2: An Example Execution of the Protected M-Process

Figure 2 illustrates an example of the execution of the protected M-Process which contains n cells. At the beginning of the execution, only cell C_0 is exposed as plain text (machine code). After the execution of some blocks in C_0, a control flow switch s_1 whose source cell is C_j occurs. Then the key corresponding to s_1 is used to XOR the whole image. It closes C_0, opens C_j, and changes all the other cells. After the encryption and decryption, the target block of s_1 is executed. At this moment, C_j becomes the only cell that is observable. When the second control flow switch s_2 happens, C_k is the target cell. The key corresponding to s_2 is used to close C_i, open C_k and modify all the others. Then the target block in C_k is executed. This process is repeated until an exit block in C_i is reached.

4.2 Encryption Key Assignment

The keys are crucial in this self-modifying scheme. An important property that the keys need to satisfy is defined as *open-close property.*

DEFINITION 4 (OPEN-CLOSE PROPERTY). *Given a program **P** which is divided into a group of cells and an encryption method **E**, let **S** be the set of all the control flow switches in **P**. A function **f** on **S** is said to have **open-close property** for **P** with encryption method **E** if the following requirements hold:*

*(i) There is a way to initialize **P** so that only the entry cell is in plain machine code.*

*(ii) After the initialization of **P**, **P** begins to execute. Every time a control flow switch **s** occurs, apply **E** on the*

whole image of $\boldsymbol{P}$ *with the key* $\boldsymbol{k = f(s)}$*. The target cell of* $\boldsymbol{s}$ *is decrypted, the source cell of* $\boldsymbol{s}$ *is encrypted and all the other cells are modified but not decrypted.*

THEOREM 4.1. *Given any program* P *and a cell partition (let* S *be the set of all the control flow switches in* P *with this cell partition), there exists at least one function* f *on* S *that satisfies the* open-close property *with encryption method XOR.*

PROOF. SKETCH: For each cell C_k, let C_k^0 be its initial state, let C_k^s be state of C_k after control flow switch s takes place. Let $sb(s)$ be the id for the source block of s, $sc(s)$ be the id for the source cell of s, $tb(s)$ be the id for the target block of s, and $tc(s)$ be id for the target cell of s.

First we build a Control Flow Graph whose nodes are the cells and interpret it as an induced undirected graph. We call this graph a Control Flow Graph in terms of Cell (CFGC). For each edge in the CFGC, a key is assigned. For any s, the value of $f(s)$ is decided by $C_{sc(s)}$ and $C_{tc(s)}$. That is, for any two control flow switches s_1 and s_2, if their source blocks are in the same cell and they also share the same target cell, the keys assigned to s_1 and s_2 are the same. We first convert the CFGC into a complete graph and then assign keys for all the edges in the graph. Let $k_{i,j}$ denote the key assigned to edge between C_i and C_j in the CFGC. Suppose there are n cells $C_0, C_1, \ldots, C_{n-1}$ in the CFGC of the M-process. We generate $n-1$ keys randomly for $k_{0,1}, k_{0,2}, \ldots, k_{0,n-1}$. All the other keys $k_{i,j}$ are derived as follows.

$$k_{i,j} = k_i \oplus k_j \qquad (4)$$

Function f is defined by

$$f(s) = k_{sc(s),tc(s)} \qquad (5)$$

, and the initialization of P is defined as follows: entry cell C_0 is not changed; any cell C_i other than C_0 is obfuscated as $C_i \oplus k_{0,i}$.

Now we are ready to show that f satisfies the two requirements listed in the definition of *open-close property*. It is obvious that the initialization mentioned above satisfies requirement (i). The main job is to prove the requirement (ii). Given any control flow switch s which occurs during the execution of the M-process at time t (right before s occurs), let $p = C_{p_1} C_{p_2} \ldots C_{p_n}$ denote the sequence of cells on the control flow path by which the program reached s from the beginning. Note that $C_{p_1} = C_0$, $C_{p_n} = C_{sc(s)}$ and there may be some duplicate cells.

At time t, every cell has been XOR'ed with a series of keys which correspond to all the control flow switches on path p. The value of XOR of all of these keys is

$$\begin{aligned}
K_s &= k_{p_1,p_2} \oplus k_{p_2,p_3} \oplus \ldots \oplus k_{p_{n-1},p_n} \\
&= k_{0,p_2} \oplus k_{p_2,p_3} \oplus \ldots \oplus k_{p_{n-1},sc(s)} \\
&= k_{0,p_2} \oplus (k_{0,p_2} \oplus k_{0,p_3}) \oplus \ldots \oplus (k_{0,p_{n-1}} \oplus k_{0,sc(s)}) \\
&= (k_{0,p_2} \oplus k_{0,p_2}) \oplus (k_{0,p_3} \oplus k_{0,p_3}) \oplus \ldots \oplus \\
&\quad (k_{0,p_{n-1}} \oplus k_{0,p_{n-1}}) \oplus k_{0,sc(s)} \\
&= 0 \oplus 0 \oplus \ldots \oplus 0 \oplus k_{0,sc(s)} \\
&= k_{0,sc(s)}
\end{aligned}$$

After applying key $f(s) = k_{sc(s),tc(s)}$ on the whole image of P, the cells are modified as follows:

1. The target cell of s

$$\begin{aligned}
C_{tc(s)}^s &= C_{tc(s)}^0 \oplus K_s \oplus k_{sc(s),tc(s)} \\
&= (C_{tc(s)} \oplus k_{0,tc(s)}) \oplus k_{0,sc(s)} \oplus k_{sc(s),tc(s)} \\
&= C_{tc(s)} \oplus k_{0,tc(s)}) \oplus k_{0,sc(s)} \oplus k_{0,sc(s)} \oplus k_{0,tc(s)} \\
&= C_{tc(s)}
\end{aligned}$$

which means that $C_{tc(s)}$ is decrypted.

2. The source cell of s is modified to

$$\begin{aligned}
C_{sc(s)}^s &= C_{sc(s)}^0 \oplus K_s \oplus k_{sc(s),tc(s)} \\
&= (C_{sc(s)} \oplus k_{0,sc(s)}) \oplus k_{0,sc(s)} \oplus k_{sc(s),tc(s)} \\
&= C_{sc(s)} \oplus k_{sc(s),tc(s)}
\end{aligned}$$

which means $C_{sc(s)}$ is encrypted.

3. For any cell C_i other than $C_{tc(s)}$ and $C_{sc(s)}$,

$$\begin{aligned}
C_i^s &= C_i^0 \oplus K_s \oplus k_{sc(s),tc(s)} \\
&= (C_i \oplus k_{0,i}) \oplus k_{0,sc(s)} \oplus k_{0,sc(s)} \oplus k_{0,tc(s)} \\
&= C_i \oplus k_{0,i} \oplus k_{0,tc(s)} \\
&= C_i \oplus k_{i,tc(s)}
\end{aligned}$$

$k_{i,tc(s)} \neq 0$ because $i \neq tc(s)$. Thus $C_i^s \neq C_i$, which means that C_i is changed but is still encrypted.

Having considered the effect of applying $f(s)$ to the whole image, we know that f satisfies all the three requirements in Equation 4 of the open-close property, which completes the proof by successfully constructing f. □

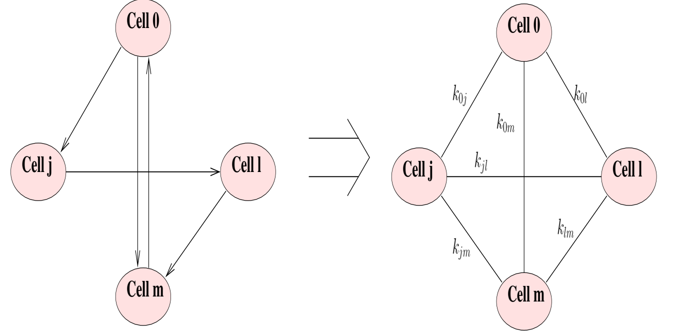

Figure 3: An Example of Key Assignment

Figure 3 shows an example of the key assignment process. The graph on the left is the original directed CFGC. It consists of four cells and five edges. After adding edges and ignoring the directions of all the edges we get the complete graph at the right. We are to generate six keys: k_{0j}, k_{0l}, k_{0m}, k_{jl}, k_{jm} and k_{lm}.

k_{0j}, k_{0l}, k_{0m} are generated randomly. The other three are calculated by equation (4). Note that the graph is undirected so there is only one key k_{0m} for the pair $< C_0, C_m >$ even though there are two edges between them in the original CFGC.

The initialization of the cells are

$$C_i^0 = \begin{cases} C_i & \text{if } i = 0 \\ C_i \oplus k_{0,i} & \text{otherwise} \end{cases}$$

4.3 Implementation Steps

In our implementation, the function `mPrcs` corresponds to the M-process. We move `mPrcs`' code from the text section to an array (called the code array) in the data section. This is because most operating systems including Linux enforce read-only access to the text section. A helper function `xorAll` performs XOR of the code image with a key. When a control flow switch s occurs during the execution of `mPrcs`, `mPrcs` calls `xorAll`. `xorAll` XOR's the whole code array, closes $sc(s)$, opens $tc(s)$, and redirects the control flow to the beginning of $tb(s)$.

Cell Partition: Given the overhead of inter-cell control flow transfer (control flow switch), the determination of cell partition for the M-process has severe repercussions on the efficiency of the scheme. Ideally, more frequently instantiated control flow edges should be absorbed within a single cell. Only the infrequent control flow edges ought to be inter-cell edges. Hence, once again, extensive profiling of the M-process CFG with SUIF is performed.

There are two parameters about the cells that need to be determined, the number of cells and their sizes. Since there is exactly one cell open at any point in time, approximately $(n-1)/n$ fraction of the code is encrypted, where n is the total number of the cells. Higher the number of cells, the more protects M-Process is from dynamic observation. However, as the number of cells increases, function `xorAll` is called more frequently, which leads to higher overhead.

Since `mPrcs` is small (it contains only twenty nine basic blocks), we analyze its CFI manually to create the cell partition. Another constraint on the cell partition is that the total size of all the blocks in each cell should be approximately the same. This is so that XOR keys can be all the same size. In such a case, much memory will be wasted by the padding at the end of the non-uniformly sized cells. One could always make XOR keys much smaller sized than the cell size (such as 32-bits). That, however, reduces the strength of XOR based encryption.

The specific cell partition for the specific M-process has six cells, each of which contains three to six basic blocks.

4.4 Function `xorAll`

The XOR operation and control redirection in the self-modifying scheme is performed by a function named `xorAll`. When a control flow switch s occurs, `mPrcs` calls `xorAll` with the key corresponding to s. `xorAll` XOR's all the cells to close the source cell and to open the target cell. There are two arguments that `xorAll` needs to help in the self-modifying scheme. The first argument is the key $k(s)$ which is used to XOR the whole code array. The second argument is the offset of $B_{tb(s)}$ in $C_{tc(s)}$. We use $off(s)$ to denote the offset. After XOR'ing the code array with $k(s)$, the whole $C_{tc(s)}$ is opened. $off(s)$ is used to calculate the starting address of $B_{tb(s)}$.

The second part of `xorAll`'s job, the control transfer, is a little tricky. After the XOR operation, the control flow is directed to $B_{tb(s)}$. Each cell starts with a *magic number* so that `xorAll` can distinguish the newly opened cell ($C_{tc(s)}$). Then `xorAll` adds the address of $C_{tc(s)}$ and offset $off(s)$ to get the address of $tc(b)$. Finally, `xorAll` modifies the return address in its stack frame, and returns so that the control flow goes to the beginning of $B_{tb(s)}$. Note that `xorAll` does not return to the end of $B_{sb(s)}$ and then execute a jump instruction from there to get to $B_{tb(s)}$. At this point $B_{sb(s)}$ is already encrypted by $k(s)$.

In-line assembly in `C` is used to generate two instructions at the end of the assembly code for `xorAll`:

```
movl -4(%ebp), %eax #code for redirection
movl %eax, 4(%ebp)  #code for redirection

leave
ret
```

In this code, `-4 (%ebp)` contains the address of the beginning of the $B_{tb(s)}$. `4(%ebp)` contains the return address for `xorAll` in the stack. After the modification of the return address, instructions `leave` and `ret` redirects the control flow to the beginning of $B_{tb(s)}$.

However, we are not done yet. We still need to guarantee a healthy stack before and after calling `xorAll`. Before a function call, all the live registers should be saved; after the call, they should be restored. In an ordinary function call, the stack cleaning and register restoring instructions follow the `call` instruction. However, after the function call to `xorAll`, the control flow will return to the beginning of $B_{tb(s)}$, not the end of the `call` instruction at the end of $B_{sb(s)}$. Thus the code to clean the arguments on the stack and restore the registers should be moved to the beginning of $B_{tb(s)}$. Also notice that $B_{tb(s)}$ may be a target block of a set S of control flow switches and $B_{tb(s)}$ is only one of them. When the control flow goes to the beginning of $B_{tb(s)}$, `mPrcs` cannot know which control flow switch in S is the one that just occurred. Because of this, the register set R saved for each control flow switch in S should be the same. Each control flow s requires the protection of a set of registers. Let $r(s)$ denote this set. To get good performance, we should analyze all the switches in s to get the union of all the sets $r(s)$. For the sake of simplicity, we save and restore all the general purpose registers and flags for every call to `xorAll`.

Basic Block Modifications: The basic blocks that are source or target blocks for control flow switches need to be modified to enable them to interact with `xorAll`. Code may be added at the beginning and at the end of a block. First let us consider the modification at the beginning of a basic block B_i. There are three cases:

1. If B_i is the entry block in `mPrcs`, no modification is needed.

2. If B_i is the target block of a control flow switch s, i.e., $i = sb(s)$, code is inserted at the beginning of B_i as follows.

```
lbl_clnStk_i:
    clean_the_stack
    restore_the_registers
lbl_thru_i:
    B_i
    ....
```

3. Else the code is modified as follows.

```
lbl_thru_i:
    B_i
    ...
```

In the pseudo-code above, `lbl_clnStk_i` is the label at the beginning of the code which cleans the stack and preserves the registers for B_i. If the control flow enters B_i from other cells, the return address of `xorAll` targets `lbl_clnStk_`i. `lbl_thru` is the label marking the beginning of the original code for B_i. It is needed because B_i may be target of

some CTI within the same cell. These in-cell CTIs should go to B_i directly without executing the code for stack cleaning and register restoration.

Another point to note is that the first block of each cell is special. If B_i is at the beginning of a cell other than Cell 0, the control-flow from B_{i-1} to B_i must go through `mPrcs` even when the last instruction of B_{i-1} is not a CTI. That's way the first block in a cell is categorized into case 2.

The modification at the end of the blocks is a little more complicated than that at the beginning. For the code at the beginning of B_i, we only consider whether B_i is a target of a control flow switch or the first block in a cell. But for the modification at the end of B_i, we need to consider the last non-directive instruction *instr* of B_i. If *instr* is a CTI, we check its target to see whether it is in the same cell or not. If the target is in another cell, we substitute *instr* by instructions which call `xorAll`. If *instr* is `ret`, nothing need to be added. If *instr* is a general-purpose instruction other than CTI (`add` and `push` for example, see [15]), we check to see whether B_i is the last block in the cell. The algorithm is described here briefly in several cases and sub-cases. Suppose B_i is in cell C_i. The code modification at the end of B_i is divided into four cases.

1. If *instr* is `ret`, then modify nothing.

2. If *instr* is `jmp target`, where *target* is in block B_k, then there are two sub cases.

   i. If B_k is also in C_i, then modify nothing

   ii. If B_k is in another cell other than C_i, then substitute *instr* with the following code

```
save_registers
pushl k_ik
pushl $0x7000000
call xorAll
```

3. If *instr* is `jcc target` (Jump if Condition Is Met such as `jc`, `jne`), where *target* is in block B_k, there are four sub cases.

   i. If B_k is also in cell C_i and B_i is the last block in C_i, then after *instr* append code to save the registers and call `xorAll`, with $k_{i(i+1)}$ as the *key* argument.

   ii. If B_k is also in cell C_i and B_i is not the last block in C_i, then after *instr* append an instruction `jmp lbl_thru_(i+1)`.

   iii. If B_k is not in cell C_i and B_i is the last block in the cell, then *instr* is substituted by the following instructions

```
    jcc lbl_svRgs_i
    save_registers
    pushl k_i(i+1)
    pushl $0x7000000
    call xorAll
lbl_svRgs_i:
    save_registers
    pushl k_ik
    pushl $0x7000000
    call xorAll
```

   iv. If B_k is in a different cell and B_i is not the last block in the cell, then substitute *instr* with the following code:

```
    jcc lbl_svRgs_i
    jmp lbl_thru_(i+1)
lbl_svRgs_i:
    save_registers
    pushl k_ik
    pushl $0x7000000
    call xorAll
```

4. If *instr* is a general-purpose instructions other than a CTI, then there are two sub-cases.

   i. If B_i is the last block in C_i, then append the following code after *instr*

```
save_registers
pushl k_i(i+1)
pushl $0x7000000
call xorAll
```

   ii. If B_i is not the last block in C_i, then after *instr*, append instruction

```
jmp l_thru_i(i+1)
```

In this algorithm, `save_registers` stands for the instructions to save the registers. There are two `pushl` before `call xorAll`. The first argument is the *key*. The immediate number `0x7000000` should be the offset of the target block in the target cell. Without generating the binary code, it is very hard for us to get the real offset of a block in the cell where the block is located. So we just push an arbitrary integer here as a placeholder. The placeholder will be substituted by the real offset when we copy the binary code to the code array.

4.5 Copying Binary Code to the Code Array

We first generate assembly code with annotations for each basic block with the help of Machine-SUIF. Once again, we ensure that all branch targets are relocatable by labeling them.

At the beginning of each cell C_i, we insert a magic number. All the binary code blocks in this cell are then copied to the code array. During this step the placeholder number `0x7000000` at the end of $B_{sb(s)}$ is substituted by the real offset of $B_{tb(s)}$ in $C_{tb(s)}$ with correct byte order (little-endian or big-endian). Finally, several `nop` instructions are appended after the last binary block in this cell. For the sake of simplicity, we use `nop` for padding. In fact, what instruction is used for padding doesn't matter for the correct result, since they will never be executed. In future work, we will put some fake blocks in the padding area to enhance obfuscation.

There is still one more difficulty in moving the binary code into the code array. We need to be careful of the global variables and functions used in `mPrcs`. If `mPrcs` uses them directly, they wouldn't be linked properly at runtime. The linker and loader won't relocate the symbols, function names or global variables in a data section. If `mPrcs` is in text section, information on the labels, function names and the global variables is placed in the relocation table. Linker and loader will find the correct address for them when the final executable is generated.

Our workaround for this problem is to call `mPrcs` as a function from `main` function which resides in the text section. Before `main` calls `mPrcs`, it initializes two arrays: one contains the pointers to all the function used by `mPrcs`; another contains pointers to all the global variables used in `mPrcs`. When `main` calls `mPrcs`, it passes the two arrays

Table 2: Space Cost Efficiency in Terms of Assembly Code

P	Source Program size (A)	Protected Program size (A)	`mPrcs` size (A)	C_s^A	$C_s'^A$
`tsort`	36k	41k	16k	0.139	0.583
`compress42`	118k	131k	16k	0.110	0.246
`test`	182k	207k	16k	0.137	0.225
`bunzip021`	210k	226k	16k	0.076	0.152

Table 3: Space Cost Efficiency in Terms of Binary Code

P	Source program size (B)	Protected program size (B)	`mPrcs` size (B)	C_s^B	$C_s'^B$
`tsort`	28k	31k	33k	0.107	1.286
`compress42`	52k	55k	33k	0.058	0.692
`test`	62k	71k	33k	0.145	0.677
`bunzip021`	72k	74k	33k	0.028	0.486

as arguments. `mPrcs` accesses all these functions and the global variables via the pointers in the arguments[1]. When the linker and loader generate the executable, they will relocate addresses for all the pointers in the two arrays, since they are located in the text section.

5. EXPERIMENTAL EVALUATION OF THE OBFUSCATOR

We used experimental methods to evaluate the efficiency of the obfuscator. The two metrics defined in definition 1, space cost $C_s(\mathcal{O}, P)$ and time cost $C_t(\mathcal{O}, P)$, are calculated for four different applications: `tsort`– topological sort from GNU core-utils; `compress42`– compress data utility; `test`– a program written by us; and `bunzip021` – an early version of bunzip utility.

For each application, we consider its space cost in terms of the assembly code and the binary code. We also calculate the cost with or without considering `mPrcs`, the M-process.

Table 2 records the space cost in terms of assembly code. P is the source program, C_s^A is the space cost without considering the size of `mPrcs` and $C_s'^A$ is the space cost including the size of `mPrcs`. 'A' stands for "in terms of assembly code".

Table 3 shows the space cost in terms of binary code. P is the source program, C_s^B is the space cost without `mPrcs` and $C_s'^B$ is the space cost inclusive of `mPrcs`. 'B' stands for "in terms of binary code".

From this data, we can draw the following two conclusions.

- The space cost both in terms of assembly code and binary code is around 1. That is, $C_s(\mathcal{O}, P) = O(1)$. This means that the space requirements of the obfuscated program are linear in the space needs of the original program.
- As the size of P increases, the relative space cost inclusive of `mPrcs` decreases. This is because the size of `mPrcs` is fixed, and the size of P-process is proportional to the space needs of the original program for a fixed shuffling rate.

[1]In fact, the format strings of the `printf` statement should also be passed as arguments in the pointer arrays

The execution time efficiency is shown in Table 4. C_t is calculated for the real run time. For each application, two types of run time are calculated. The first run time is for P-process working with a non-obfuscated M-process (we coded a function to simulate `mPrcs` without the self-modifying scheme). The second run time is for P-process working with the obfuscated M-process. This separates the overhead incurred by the CFI-hiding scheme from the overhead of the self-modifying scheme.

This data indicates the time cost to be less than 10. This is with the shuffling rate of 20%, which is pretty high. When the shuffling rate is set to 5%, the time overhead for the four applications is shown in Table 5. Take `tsort` as an example. The time cost drops 83.2% with plain M-process and 61% with obfuscating M-process. With the small shuffling rate, all the time costs are less than 4, which is acceptable for most programs.

The relative overhead of the self-modifying scheme and the CFI-hiding scheme seems to indicate that the self-modified code based obfuscation costs more than the basic CFI-hiding scheme. This is because the modification of the entire M-process through XOR is pretty expensive. It is even more expensive than IPC, the main overhead in the CFI-hiding scheme.

6. CONCLUSIONS AND FUTURE WORK

We presented an implementation of a two-process obfuscation scheme. The static layout of the program process (P-process) is randomly permuted from the original program's static layout at the boundaries of hot nodes – a group of adjacent basic blocks. The permutation itself is stored in jump table which is part of another process, monitor process (M-process). M-process is forked as a co-process of P-process. Control flow is normal within a hot node of P-process. When a permuted instruction is encountered, P-process communicates with the M-process to perform an address translation through jump-table lookup. M-process is a small program with a small wait loop waiting on jump-table lookup requests from P-process. This allows for more extensive protection for the M-process. Specifically, we use a variant of Aucsmith's scheme based on self-modifying code to hide/obfuscate M-process. The entire system was implemented with gcc and machineSUIF combination. We present both the space and run time overhead data for a collection of programs.

The main future work will involve reducing the cost of inter-process communication between P-process and M-process. Specifically, use of prefetching the address translations so as to hide the latency of lookup can be explored.

7. REFERENCES

[1] David Aucsmith. (1996). *Tamper Resistant Software: An Implementation*, Proceedings of the First International Workshop on Information Hiding.

[2] Business Software Alliance. (2005). *Second Annual BSA and IDC Global Software Piracy Study*, http://www.bsa.org/globalstudy/upload/2005-Global-Study-English.pdf, May 2005.

[3] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, Ke Yang. (2001). *On the (Im)possibility of Obfuscating Programs.* Lecture Notes in Computer Science Vol 2319.

Table 4: Run Time Overhead (shuffling rate: 20%)

P	Running Time of Not Protected Programs			Running Time of PP With an un-protected MP				Running Time of PP With Protected MP			
	real	user	sys	real	user	sys	C_t	real	user	sys	C_t
`tsort`	4.90	0.29	0.05	9.85	1.60	1.05	1.01	18.45	1.61	1.12	2.77
`compress42`	2.05	0.42	0.52	10.31	2.31	2.77	4.03	20.22	3.30	3.10	8.87
`test`	34.33	2.45	1.40	60.22	6.19	4.23	0.75	150.10	6.20	5.89	3.37
`bunzip021`	6.46	0.62	0.54	25.72	7.33	5.41	2.98	42.42	7.20	6.63	5.57

Table 5: Time Cost Efficiency (shuffling rate: 5%)

P	Running Time of Not Protected Programs			Running Time of PP With a un-protected MP				Running Time of PP With Protected MP			
	real	user	sys	real	user	sys	C_t	real	user	sys	C_t
`tsort`	4.90	0.29	0.05	5.73	0.65	0.29	0.17	10.20	0.66	0.31	1.08
`compress42`	2.05	0.42	0.52	3.58	0.52	0.75	1.56	7.933	1.46	1.55	2.87
`test`	34.33	2.45	1.40	42.57	4.06	2.36	0.24	150.10	6.20	5.89	3.37
`bunzip021`	6.46	0.62	0.54	11.18	1.44	1.89	0.73	20.16	2.20	2.33	2.12

[4] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. (2001). *An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs*. In G.I. Davida and Y. Frankel, editors, ISC 2001, Lecture Notes in Computer Science 2200, pages 144–155. Springer-Verlag.

[5] Christian Collberg, Clark Thomborson, Douglas Low. (1997). *A Taxonomy of Obfuscating Transformations*. Technical Report 148, Department of Computer Science, University of Auckland.

[6] Christian Collberg, Clark Thomborson, Douglas Low. (1998). *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs*. Symposium on Principles of Programming Languages, pp. 184-196.

[7] Christian Collberg, Clark Thomborson, Douglas Low. (1998). *Breaking Abstractions and Unstructuring Data Structures*. IEEE International Conference on Computer Languages, Chicago, IL.

[8] Christian Collberg, Clark Thomborson. (2000). *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection*. Technique Report 170, Department of Computer Science, the University of Auckland.

[9] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. (2001). *Introduction to Algorithms*. MIT and McGraw-Hill.

[10] S. Forrest, A. Somayaji, D. Ackley. (1997). *Building Diverse Computer Systems*. Proceedings of 6th Workshop on Hot Topics in Operating Systems, Computer Society Press, Los Alamitos, CA.

[11] Michael R. Garey and David S. Johnson. (1979). *Computers and Intractability*. W. H. Freeman, New York.

[12] Glenn Holloway, Cliff Young. (1997). *The Flow Analysis and Transformation Libraries of Machine SUIF*. Proc. Second SUIF Compiler Workshop.

[13] J. Hennessy, D. Patterson. (1990). *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann.

[14] Steven Homer, Alan L. Selman. (2001). *Computability and Complexity Theory*. Springer Verlag New York.

[15] *IA-32 Intel Architecture Software Developer's Manual*

[16] Richard M. Karp. (1972). *Reducibility among combinatorial problems*. Complexity of Computer Computations, pp. 85-103, Plenum Press.

[17] D. E. Knuth. (1997) *The Art of Computer Programming, Volume 2, Third edition. Section 3.4.2, Algorithm P, pp 145*. Reading: Addison-Wesley. ISBN: 0-201-89684-2.

[18] Michael D. Smith. (1996). *Extending SUIF for Machine-dependent Optimizations*. Proc. First SUIF Compiler Workshop.

[19] Eelco Visser. (2001). *Survey of Strategies in Program Transformation Systems*. Volume 57 of Electronic Notes in Theoretical Computer Science, Department of Computer Science, the University of Auckland.

[20] Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson. (2000). *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Technical Report CS-2000-12, University of Virginia.

[21] Chenxi Wang. (2000). *A Security Architecture for Survivability Mechanisms*. Ph.D. Dissertation, Department of Computer Science, University of Virginia.

[22] Gregory Wroblewski. (2002). *General Method of Program Code Obfuscation*. PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics.

[23] Gregory Wroblewski. (2002). *General Method of Program Code Obfuscation*. Proceedings of the International Conference on Software Engineering Research and Practice (SERP) 2002, Las Vegas, USA.

[24] Cliff Young. (1996). *The SUIF Control Flow Graph Library*. Harvard University, Cambridge, MA.

[25] Vladimir Zakharov. (2001). *Preliminary Analysis of the Security of Control Flow Code Transformations*. Cloakware & Code Transformations Report, Cloakware and the Institute of Systems Programming of the Russian Academy of Science.

[26] Xiangyu Zhang, Rajiv Gupta, Youtao Zhang. (2003). *Hiding Program Slices for Software Security* Intl. Symp. on Code Gen. and Opt.