An approach to the obfuscation of control-flow of sequential computer programs

Stanley Chow[1], Yuan Gu[1], Harold Johnson[1], and Vladimir A. Zakharov[2,3]

[1] Cloakware Corporation, Kanata, Ontario, Canada K2L 3H1
(stanley.chow,yuan.gu,harold.johnson)@cloakware.com
[2] Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow, RU-119899, Russia
zakh@cs.msu.su
[3] Institute for System Programming, Russian Academy of Sciences, B. Kommunisticheskaya, 25, 109004 Moscow, Russia

Abstract. In this paper we present a straightforward approach to the obfuscation of sequential program control-flow in order to design tamper-resistant software. The principal idea of our technique is as follows: Let I be an instance of a hard combinatorial problem C, whose solution K is known. Then, given a source program π, we *implant* I into π by applying semantics-preserving transformations and using K as a key. This yields as its result an obfuscated program $\pi_{I,K}$, such that a detection of some property $\mathcal{P}$ of $\pi_{I,K}$, which is essential for comprehending the program, gives a solution to I. Varying instances I, we obtain a family Π_C of obfuscated programs such that the problem of checking $\mathcal{P}$ for Π_C is at least as hard as C. We show how this technique works by taking for C the acceptance problem for linear bounded Turing machines, which is known to be PSPACE-complete.

1 Introduction

One of the most significant achievements in cryptographic research in recent years has been to establish complexity-theoretic foundations for most classical cryptographic problems. This makes it possible to develop new methods for encryption, authentication, and design of cryptographic protocols using a solid framework for estimating their resistance to attack. However, there remain some important problems in cryptography whose theoretical foundations are still rather weak.

One such problem is: how can we create tamper-resistant software (TRS). A program converted to TRS form has the property that understanding and and making purposeful modifications to it, are rendered difficult, while its original functionality is preserved. Such

TRS is very important when it is necessary to ensure the intended operation of a program and to protect its secret data and algorithms in a potentially hostile environment. The difficulty is that any program presents the *same* information (namely, an executable embodiment) to an authorized user of the program, and to an adversary seeking to extract its secrets or modify its behavior. The difference between licit and illicit use is the way in which the program is employed. An authorized user is interested only in correct executions of the program. To achieve this the program should supply a processing device only with 'local' information: at every state of a run it has to determine which instruction to be performed currently, what data it is applied to, and at what state a control to be passed next. An adversary, on the other hand, seeks to extract 'global' knowledge from the program, such as relationships between variables, intended meaning of data structures, parameters, routines and algorithms used in a program, etc. The only way to obtain the 'global' information is to study behavior of the program by means of static and statistical analysis tools. Thus, to hamper this activity, the program should be presented in a form which hinders its global comprehension as much as possible.

Tentative research on constructing TRS has been initiated in [3, 4, 17, 21]. The key idea offered in these papers is that of developing a *program obfuscation* technique. Informally, program obfuscation is any semantics-preserving transformation of a source computer program which performs deep and sophisticated changes in its control-flow and data-flow in order to make a target program 'unreadable' while preserving its functionality. This can be achieved by applying some equivalent transformations to a program, such as replacing and shuffling operational codes, inserting dead and irrelevant codes, data encoding, etc. A wide variety of obfuscating transformations of this kind is presented in [3]. Some of them have been successfully implemented in a number of projects aimed at strengthening security of Java software (see [9, 19, 20]). While these transformations look quite useful and natural, all share the same principal shortcoming: a lack of any theoretical foundations which guarantee their obfuscating effectiveness.

In [21], an attempt is made to estimate a resistance of an *aliasing* technique. The introduction of aliasing into a program by means of arrays or pointers is intended to restrict the precision of static data-flow analysis. In [8, 16, 21] it was shown that many static analysis problems involving alias detection are NP-hard. This is shown by reducing the 3-SAT problem to that of determining indirect targets in the presence of aliasing. But when studying the proofs of these assertions one can readily see that the reduction methods may work in more general cases. This is due to the very nature of many computation models which enables us to embody many kinds of combinatorial problems in program control-flow and data-flow structures.

Relying on such considerations we offer the following strategy aimed at impeding static analysis of computer programs. Suppose I is an instance of a hard combinatorial problem C, whose solution K is known. Then, given a source program π, we *implant* I into π by applying semantics-preserving transformations and using K as a key. This yields as the result an obfuscated program $\pi_{I,K}$, such that detection of some essential property $\mathcal{P}$ of $\pi_{I,K}$ which is necessary for comprehending the program gives a solution to I. Varying instances I, we get a family Π_C of obfuscated programs, such that the problem of checking $\mathcal{P}$ for Π_C is at least as hard as C. At the same time everyone who knows a key K can easy reveal $\mathcal{P}$ for $\pi_{I,K}$. Thus, K may be considered as a *watermark* of $\pi_{I,K}$ whose resistance is guaranteed by the hardness of C.

In this paper we demonstrate how to apply this approach to the obfuscation of control-flow for sequential computer programs. In section 2 we describe preliminary transformations of sequential programs that *flatten* their control-flow structure. These transformations were developed at Cloakware (see [2, 20] for more details) in order to convert computer programs to a form highly amenable to further obfuscation techniques. Therefore when referring to this flattening machinery we will call it *cloaking technology.* A control-flow of a source program is grouped on a switch statement called a *dispatcher*, so that the targets of *goto* jumps are determined dynamically. A dispatcher may be viewed as a deterministic finite-state automaton realizing the overall control over a flattened program. Hence, to obfuscate the program control-flow *per se*, it suffices to focus on the dispatcher only. In section 3 we recall the concept of a *linear bounded Turing machine* (LBTM)[15] and show thereafter that the acceptance problem for LBTMs is LOGSPACE reducible to the reachability problem for cloaked program dispatchers: the problem of checking if there exists a run that transfers a dispatcher from the initial state q_0 to some specific state q_1. Since the acceptance problem for LBTMs is known to be PSPACE-complete[6], this implies that the reachability problem for flattened program dispatchers is PSPACE-hard. After considering in section 5 some basic properties of sequential programs that are essential for their comprehension and further global manipulations, we demonstrate how to implant the acceptance problem for an arbitrary LBTM M into any dispatcher D in order to hamper the detection of these properties. As a result we obtain obfuscated programs whose control-flow is protected from those tampering attacks which are based on static analysis. The resistance of the obfuscation technique is guaranteed by the PSPACE-hardness of combinatorial problems to be solved in attempting to detect some essential properties of program control-flow. We are sure that the same implantation technique is applicable to the obfuscation of many control-flow and data-flow properties of sequential programs. In practice it is advisable to strengthen this approach by implanting into programs a number of combinatorial problems of different types, to obviate security breaches using special-purpose static analyzers.

2 Flattening program control-flow

For the sake of simplicity, and in order to emphasize the versatility of our approach, we restrict our consideration to sequential programs whose syntax includes only simple variables and operations, and labelled statements of the following form:

- *assignment instructions* `x` $\leftarrow$ `t`, where `x` is a variable (integer or boolean) and `t` is an arithmetic expression or a predicate;
- *input instructions* `READ(x)`;
- *output instructions* `WRITE(x)`;
- *control transfer instructions* `COND b,` $\mathtt{l}_1$`,` $\mathtt{l}_2$ and `GOTO` $\mathtt{l}_1$, where `b` is a boolean variable and $\mathtt{l}_1$`,` $\mathtt{l}_2$ are labels of statements;
- *exit instructions* `STOP`.

These statement forms have their conventional semantics. A program is a sequence of labelled statements. A *basic block* is a sequence of input, output and assignment instructions which ends in `COND` $\mathtt{l}_1$`,` $\mathtt{l}_2$, `GOTO` $\mathtt{l}_1$, or `STOP` instructions and is executed strictly from the first to the last statement. Basic blocks cannot contain control transfer instructions except as the last statement. We denote by

$Cond$ the set of boolean variables $b_1, b_2, \ldots, b_M$ occurring in `COND` instructions, and by Σ the set of all possible tuples of binary values of these variables.

A cloaking transformation of a program consists of several steps. Briefly, they are as follows (considerably simplified; see [2] for more details):

1. **Splitting basic blocks into pieces.** Each basic block is split into several pieces. A *piece* is a sequence of instructions executed strictly from the first to the last instruction; i.e., each piece is part of a basic block. The same block may be split into pieces many different ways. Several copies of the same piece are also possible.
2. **Introducing dummy pieces.** In this step some *faked* pieces meant for obscuring useful operations are added. We denote by $P_1, P_2, \ldots, P_r$ all pieces (genuine and dummy) introduced so far. Each piece is tagged individually. The set of all *tags* is denoted by Tag. Every piece P_i, except ones that end in `STOP` instruction, may has several successors. Its successors are determined by values of boolean conditions $b_1, b_2, \ldots, b_M$: genuine conditions are used for branching, whereas faked conditions are used for the simulation of nondeterministic choice between similar pieces. A *control function* $\Phi : Tag \times \Sigma \rightarrow Tag$ is defined for selection of successors.
3. **Variable renaming.** For each piece in the set of pieces, all variables used in the piece are renamed to names which are unique. As a consequence each piece will operate over its own set of variables. In this step an internal renaming table Tab is produced which associates the old names of variables with new ones.
4. **Connective lump forming.** For every pair of pieces P_i, P_j that result from modification at the previous step a *connective lump* is generated. A connective lump is a sequence of move instruction of the form $\texttt{x} \leftarrow \texttt{y}$. It is destined to conform variables used in P_i with ones occurred in P_j. Move instructions are generated by the table Tab. Connective lumps $LC_1, LC_2, \ldots, LC_k$ are marked with individual labels $lc_1, lc_2, \ldots, lc_k$. This set of labels is denoted by $LabC$.
5. **Emulative lump forming.** In this step a set of *emulative lumps* is formed from the set of pieces. An emulative lump is composed of several pieces merged together. Each piece may appear only

in a single lump, and all pieces must be employed in the lumps. Actually, an emulative lump looks like a basic block. The only difference is that every time when an emulative lump is executed only a single piece influences upon the computation; intermediate results computed by other pieces are discarded. A piece whose intermediate results are retained for further computations is determined dynamically. Emulative lumps $LE_1, LC_2, \ldots, LE_n$ are marked with individual labels $le_1, le_2, \ldots, le_n$. This set of labels is denoted by $LabE$. The product $Tag \times LabE \times LabC$ is denoted by Δ

6. **Dispatcher lump forming.** In the previous steps the basic blocks for cloaked program are formed. However, they are still connected with control transfer instructions. To obscure explicit control transference a *dispatcher lump* D is added in the beginning of a flattened program. A dispatcher evaluates control function Φ and jumps either to the emulative lump whose piece to be executed next, or to the connective lump to join successive pieces. A dispatcher may be thought of as a deterministic finite automaton (DFA) whose operation is specified by its output function
$$\Psi :\ Tag \times \Sigma \ \rightarrow \ \Delta,$$
which for every piece P_i and a tuple σ of values of boolean conditions yields the triple $\Psi(tag_i, \sigma) = (tag_j, le, lc)$, such that the piece P_j tagged with $tag_j = \Phi(tag_i, \sigma)$ is the successor of P_i, le is the label of an emulative lump LE containing P_j, and lc is the label of a connective lump LC which joins P_i and P_j. Dispatcher is implemented as a switch statement composed of control transfer instructions.

As seen from the above, a cloaked program is composed of three main parts: emulative lumps, connective lumps, and a dispatcher. Computing operations are grouped on emulative lumps. To obscure this part it is useful to apply algebraic and combinatorial identities and/or secret sharing techniques [1]. Cloaked program data-flow is assigned on connective lumps. It can be obfuscated by applying data encoding techniques [5]. We focus on the obfuscation of flattened program control-flow which is managed by a dispatcher.

Clearly, a dispatcher is the key component of a cloaked program: without means for analyzing a dispatcher, one can not get any reasonable knowledge about program behavior. A dispatcher, viewed as a finite automaton, is easy to comprehend when its state space is rather small. Therefore, to hamper the analysis of a cloaked program control-flow, we to expand enormously the state-spaces of dispatchers. But even with such expansion, there still exists a possible threat of de-obfuscation by means of some minimization technique for finite automata. In the next sections we demonstrate how to reduce the effectiveness of minimization attacks by implanting instances of hard combinatorial problems into dispatchers.

3 The acceptance problem for LBTMs

Linear bounded Turing machines (LBTMs) were introduced in [15]. An LBTM is exactly like a one-tape Turing machine, except that the input string x is enclosed in left and right end-markers $\vdash$ and $\dashv$ which may not be overwritten. An LBTM is constrained never to move left of $\vdash$ or right of $\dashv$, but it may read and write arbitrarily between the end-markers in the way which is usual for a conventional Turing machine.

Formally, an LBTM is an octuple $\langle A, B, \vdash, \dashv, S, s_0, s_a, T\rangle$, where

– A and B are the input and the alphabets, respectively;
– $\vdash, \dashv$ are the endmarkers;
– S is the set of states, s_0 is the start state, and s_a is the accepting state;
– T is the program which is a set of pentuples

$$T \subseteq (B \cup \{\vdash, \dashv\}) \times S \times \{L, R\} \times B \times S$$

such that no pentuples of the form $(s, \vdash, L, b, q')$, $(s, \dashv, R, b, s')$, $(s, \vdash, R, d, q')$, $(s, \dashv, L, e, q')$, where $d, e \neq \dashv$, are admissible in T.

Every pentuple in T is called a *command*. LBTM M is called *deterministic* if for every pair $b \in B, s \in S$, no two different commands begin with the same prefix b, s. In what follows only deterministic LBTMs are considered.

Let $w = b_1 b_2 \dots b_n$ be a word over B and M be an LBTM. Then a configuration of M on w is any word of the form

$$\vdash b_1 b_2 \dots b_{i-1} s b_i b_{i+1} \dots b_n \dashv$$

This configuration will be denoted by (w, s, i) assuming that $(w, s, 0)$ and $(w, s,\ n+1)$ correspond to $s \vdash b_1 b_2 \dots b_n \dashv$ and $\vdash b_1 b_2 \dots b_n s \dashv$

respectively. The application of a command to a configuration is defined as usual (see [12]). Given a configuration α we denote by $T(\alpha)$ the set of configurations that are the results of applications of all possible commands in T to α. A run of an LBTM on an input word $w \in A^*$ is a sequence (finite or infinite) of configurations

$$\alpha_0, \alpha_1, \ldots, \alpha_n, \alpha_{n+1}, \ldots ,$$

such that $\alpha_0 = (w, s_0, 1)$ and for every n, $n \geq 1$, α_{n+1} is in $T(\alpha_n)$. A run is called *accepting* iff $\alpha_n = (w', s_a, i)$ for some n (recall that s_a is the accepting state). We say that a LBTM M *accepts an input word* $w \in A^*$ iff the run of M on w is accepting. The set of all inputs accepted by M is denoted by $L(M)$. The *acceptance problem* for LBTMs is to check given LBTM M and an input word w whether w is in $L(M)$.

It is known that the acceptance problem for LBTMs, namely a language ACCEPT $= \{(w, M) : \ w \in L(M)\}$, is PSPACE-complete[6]. In the next sections we prove that the acceptance problem for LBTMs is reducible to the reachability problem for flattened program dispatchers.

4 The reachability problem for dispatchers

Formally, a deterministic finite automaton associated with a dispatcher D of a flattened program is a sextuple

$$D = \langle \Sigma, \Delta, Q, q_0, \varphi, \psi \rangle,$$

where

- Σ and Δ are the input and output alphabets of D, respectively;
- Q is the set of internal states, and q_0 is the initial state;
- $\varphi : \ Q \times \Sigma \ \rightarrow \ Q$ is the transition function;
- $\psi : \ Q \ \rightarrow \ \Delta$ is the output function.

We assume that both of the alphabets Σ and Δ are encoded in binary. Then the transition and output functions are boolean operators that can be implemented by means of boolean expressions (formulae) over some conventional set of boolean connectives (operations), say $\vee$, $\neg$, etc. The total size of all boolean formulae involved in the specification of D is denoted by $|D|$.

Given a dispatcher D, we extend its transition function φ on the set of all finite words Σ^* over the input alphabet Σ by assuming $\varphi^*(q, \varepsilon) = q$ for the empty word ε, and $\varphi^*(q, w\sigma) = \varphi(\varphi^*(q, w), \sigma)$ for every word w in Σ^* and tuple σ in Σ. We say that a state q' is *reachable* from a state q iff $q' = \varphi^*(q, w)$ holds for some input sequence (word) w from Σ^*. The *reachability problem* for dispatchers is to check for a given dispatcher D, its internal state q, and a set of internal states Q', whether some state q', $q' \in Q'$ is reachable from q in D.

To prove PSPACE-completeness of the reachability problem we show at first that it is decidable in polynomial space and then demonstrate that ACCEPT is LOGSPACE-reducible to the reachability problem.

Theorem 1. *The reachability problem for dispatchers is in* PSPACE*.*

Proof. The reachability of a state q' from a state q in some dispatcher D specified in terms of boolean formulae can be recognized by means of a well-known dichotomic search (see [18]): to check that q' is reachable from q in less than 2^n steps it is suffice to cast some intermediate state q'' and then check by applying the same procedure recursively that both q'' and q' are reachable from q and q', respectively, in less than 2^{n-1} steps. QED

To show that the reachability problem is PSPACE-complete, we will restrict our consideration to the dispatchers of some specific type. A dispatcher D is called *autonomous* if its transition function φ does not depend on inputs, i.e. $\varphi(q, \sigma_1) = \varphi(q, \sigma_2)$ holds for each state q and every pair σ_1, σ_2 of inputs.

Theorem 2. *For every input word w and* LBTM *M there exist an autonomous dispatcher D, a state q_0, and a set of states Q', such that M accepts w iff some q_1, $q_1 \in Q'$ is reachable from q_0 in D.*

Proof. Without loss of generality, both alphabets A and B for M are assumed to be binary. Suppose that $|w| = n$ and M has $|S| = 2^m$ states. We encode each state s in S by binary tuple $\gamma_s = \langle d_1, \ldots, d_m \rangle$ and introduce three sets of boolean variables

$$\begin{aligned} X &= \{x_1, x_2, \ldots, x_n\}, \\ Y &= \{y_0, y_1, y_2, \ldots, y_n, y_{n+1}\}, \\ Z &= \{z_1, z_2, \ldots, z_m\}, \end{aligned}$$

for encoding contents of linear bounded tape of M, positions of the tape, and the states of M. Namely, every configuration (w', s, i) is encoded by the tuple $\langle \hat{x}_1, \dots, \hat{x}_n, \hat{z}_1, \dots, \hat{z}_k, \hat{y}_0, \hat{y}_1, \dots, \hat{y}_{n+1}\rangle$, such that $\hat{x}_1 \dots \hat{x}_n = w'$, $\langle \hat{z}_1, \dots, \hat{z}_m\rangle$ is the code of s, and $\langle \hat{y}_0, \hat{y}_1, \dots, \hat{y}_n, \hat{y}_{n+1}\rangle$ contains exactly one 1 at the position i. Since M is deterministic, for every command beginning with a pair a, s we denote by $b_{a,s}$ the tape symbol to be written instead of a, by $\gamma_{a,s}$ the code of the state M has to pass to by the command, and by $w_{a,s}$ the indication of the direction M has to move its head by the command (i.e. $w_{a,s}$ is 0 if the head has to be move to the left, and 1 if it has to be move to the right).

Now we specify an autonomous dispatcher $D_{w,M}$ which simulates the run of M on w. Consider the following boolean formulae

$$\begin{aligned}
f(\bar{x}, \bar{y}) &= \bigvee_{i=1}^{n} (x_i \wedge y_i),\\
g_\omega(\bar{z}) &= \bigwedge_{i=1}^{k} z_i^{a_i}, \text{ for every } \omega = \langle a_1, a_2, \dots, a_k\rangle\\
F_i(\bar{x}, \bar{y}, \bar{z}) &= x_i \vee (y_i \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge b_{a,s})),\\
1 \le i \le n,&\\
G_j(\bar{x}, \bar{y}, \bar{z}) &= \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge \gamma_{a,s}[j]),\\
1 \le j \le m,&\\
H_k(\bar{x}, \bar{y}, \bar{z}) &= y_{i+1} \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge \neg w_{a,s}) \vee\\
&\quad y_{i-1} \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge w_{a,s}),\\
1 < k < n,&\\
H_1(\bar{x}, \bar{y}, \bar{z}) &= y_2 \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge \neg w_{a,s}) \vee y_0,\\
H_n(\bar{x}, \bar{y}, \bar{z}) &= y_{n+1} \vee y_{n-1} \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge w_{a,s}),\\
H_0(\bar{x}, \bar{y}, \bar{z}) &= y_1 \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\bar{z}) \wedge (f(\bar{x}, \bar{y}) \equiv a) \wedge \neg w_{a,s}),\\
H_{n+1}(\hat{x}, \hat{y}, \hat{z}) &= y_n \wedge \bigvee_{s\in S} \bigvee_{a\in\{0,1\}} (g_{\gamma_s}(\hat{z}) \wedge (f(\hat{x}, \hat{y}) \equiv a) \wedge w_{a,s}),
\end{aligned}$$

where notation x^a stands for x when $a = 1$, and for $\neg x$ when $a = 0$. It is easy to notice that the size of every formula above is $O(|w||S|\log|S|)$ and all these formulae may be constructed effectively by some Turing machine which operates in space logarithmic in $|S| + |w|$.

The formulae F_i, G_j, and H_k specify the rewriting actions, the transition actions, and the moving of the LBTM's head. It is a matter of direct verification to prove that whenever $\langle \hat{x}, \hat{y}, \hat{z} \rangle$ encodes some configuration α of LBTM M then $\langle \bar{F}(\hat{x}, \hat{y}, \hat{z}), \bar{G}(\hat{x}, \hat{y}, \hat{z}), \bar{H}(\hat{x}, \hat{y}, \hat{z}) \rangle$ stands for the configuration $T(\alpha)$, where T is the program of M.

A required autonomous dispatcher $D_{w,M} = \langle \Sigma, \Delta, Q, q_0, \varphi, \psi \rangle$ is one whose state space Q is the set $\{0,1\}^{2n+k+2}$ of all possible binary tuples of the length $2n+k+2$, the initial state is the tuple $\langle \bar{x}_0, \bar{y}_0, \bar{z}_0 \rangle$, such that $\hat{x}_0 = w$, $\hat{y} = 010\dots0$, $\hat{z} = \gamma_{s_0}$, and transition function is specified by the boolean operator $\langle \bar{F}, \bar{G}, \bar{H} \rangle$. Then, by the construction of these formulae, LBTM M accepts w iff some state $\langle \bar{x}, \bar{y}, \bar{z}_a \rangle$, such that $\hat{z}_a = \gamma_{s_a}$, is reachable from the initial state.

Thus, the acceptance problem for LBTM M is reduced to the reachability problem for cloaked program dispatcher $D_{w,M}$. QED

5 Redundancy-checking for cloaked programs

Most methods of static data-flow and control-flow analysis[7, 11, 13] compute their solutions over paths in a program. As applied to cloaked programs paths are defined as follows. Let π be a cloaked program composed of a dispatcher $D = \langle \Sigma, \Delta, Q, q_0, \varphi, \psi \rangle$, a set of emulative lumps, and a set of connective lumps. Given a sequence of $w = \sigma_1, \sigma_2, \dots, \sigma_n$ of tuples from Σ we say that the sequence instructions formed out lumps

$$LE_1, LC_1, LE_2, LC_2, \dots, LE_n, LC_n \tag{1}$$

is a path iff this sequence meets the following requirements:

1. $\psi(\varphi^*(q_0, \sigma_1\sigma_2\dots\sigma_i)) = (tag_i, le_i, lc_i)$ for every i, $1 \leq i \leq n$;
2. emulative lumps $LE_1, LE_2, \dots, LE_{n-1}$ do not terminate program runs, i.e. they have no `STOP` statements;
3. an emulative lump LE_n terminates the program.

We denote a sequence (1) by $path(\pi, w)$. By the result $[path(\pi, w)]$ of (1) we mean the sequence of tuples of terms that stand for the arguments in the predicates and the output statements that are checked and executed along the path. It is easy to see that every feasible run of π can be associated with its path for some appropriate sequence w, whereas the opposite is not true in general. Two programs π_1 and π_2 having the same set of predicates are called *path-equivalent* iff $[path(\pi_1, w)] = [path(\pi_2, w)]$ for every sequence of w of tuples from Σ. It should be noticed (see [14, 22]) that path-equivalent programs compute the same function (input-output relation), i.e., path-equivalence approximates functional equivalence for sequential programs.

We say that

- an emulative lump LE is *dead* in a program π iff no paths in π contain LE.
- an instruction s is *faked* in a program π iff by removing s from π we get a program π' which is path-equivalent to π.
- a variable x is *superfluous* in a program π if by replacing every occurrence of x in π with some constant and removing all assignments whose left-hand side is x we obtain a program π' which is path-equivalent to π.

Intuitively, dead lumps and faked instructions are those which do not influence the program input-output behavior and, hence, can be removed without loss of program correctness. In what follows, by *redundancy problems* we mean the problems of checking for dead lumps, faked instructions, and superfluous variables in programs.

The redundancy of program components is the basic property to be checked to comprehend (or to optimize) a program. Therefore, it is highly reasonable to measure a resistance of obfuscated programs in terms of the complexity of redundancy-checking for these programs. When the dispatcher of a cloaked program is implemented explicitly (say, by tableaux), the redundancy problem (w.r.t. path-equivalence) is decidable in polynomial time[13, 14]. In the next section we prove that the redundancy-checking for cloaked programs is PSPACE-hard when dispatchers of cloaked programs are implemented implicitly by means of boolean formulae.

6 PSPACE-hardness of cloaked program analysis

We show that the above redundancy problems for cloaked programs are PSPACE-complete. This is achieved through the implantation of instances of the acceptance problem for LBTMs into an arbitrary dispatcher. The implantation technique makes it possible to reduce the acceptance problem for LBTMs to many important static analysis problems for cloaked programs. A similar method was used in [10] for proving PSPACE-hardness of some analysis problems for simple programs.

Theorem 3. *Let π be an arbitrary cloaked program, D be a dispatcher of π, and w be some input word for* LBTM *M. Then π can be transformed to a cloaked program $\pi_{w,M}$ which meets the following requirements:*

1. *the description length and running time of $\pi_{w,M}$ are at most linearly larger than that of π, w, and M;*
2. *$\pi_{w,M}$ is path-equivalent to π iff M does not accept w.*
3. *$\pi_{w,M}$ contains a distinguished emulative lump LE_0 which consists of a single instruction* `y` $\leftarrow$ `0`*, such that LE_0 is dead and* `y` *is superfluous iff M does not accept w.*

Proof. For simplicity we will assume that π contains a single output instruction `WRITE(x)`. Let `y` be a variable which does not occur in π. The desired program $\pi_{w,M}$ results from π through the following transformations:

1. An assignment `y` $\leftarrow$ `0` is added to the entry lump whose execution begins every run of π;
2. An assignment `x` $\leftarrow$ `x + y` is inserted immediately before the output instruction;
3. An emulative lump LE_0 which consists of a single piece P_0: `y` $\leftarrow$ `1`, and an empty connective lump LC_0 are introduced; these lumps are labelled with le_0 and lc_0, respectively;
4. The dispatcher D' is as follows. Let $D = \langle \Sigma, \Delta, Q^\pi, q_0^\pi, \varphi^\pi, \psi^\pi \rangle$ be a dispatcher of π, and $D_{w,M} = \langle \Sigma, \Delta, Q^M, q_0^M, \varphi^M, \psi^M \rangle$ be an autonomous dispatcher corresponding to the acceptance problem for w and M as it was shown in Theorem 2. Denote by Q_a^M the set of those states in $D_{w,M}$ that indicate the acceptance of configurations by M. Then $D' = \langle \Sigma, \Delta, Q', q_0', \varphi', \psi' \rangle$, where $Q' = Q^\pi \times Q^M \times \{0,1\}$, $q_0' = \langle q_0^\pi, q_0^M, 0 \rangle$, and for each state $q = \langle q^\pi, q^M, \xi \rangle$ in Q'

$$\varphi'(q,\sigma) = \begin{cases} \langle \varphi^\pi(q^\pi,\sigma), \varphi^M(q^M,\sigma), \xi \rangle, \text{ if } q^M \notin Q_a^M \text{ or } \xi = 1, \\ \langle q^\pi, q^M, 1 \rangle, \text{ if } q^M \in Q_a^M \text{ and } \xi = 0, \end{cases}$$

$$\psi'(q,\sigma) = \begin{cases} \psi^\pi(q^\pi,\sigma), \text{ if } q^M \notin Q_a^M \text{ or } \xi = 1, \\ \langle P_0, le_0, lc_0 \rangle, \text{ if } q^M \in Q_a^M \text{ and } \xi = 0, \end{cases}$$

It immediately follows from the construction of $\pi_{w,M}$ that the emulative lump LE_0 appears in some path of $\pi_{w,M}$ iff M accepts w. It follows therefrom that $\pi_{w,M}$ satisfies the requirements above. QED

Corollary 1. *Redundancy problems for cloaked programs are* PSPACE*-hard.*

Corollary 2. *Minimization of cloaked program dispatchers is* PSPACE*-hard.*

7 Conclusions and Acknowledgments

We have presented an approach to designing tamper-resistant software where an obfuscation of program control-flow is achieved by implanting instances of hard combinatorial problems into programs. The tamper-resistance of our obfuscation technique is guaranteed by the hardness of problems an adversary would have to solve when attempting to detect the essential properties of obfuscated programs through their static analysis.

We would like to thank the anonymous referee for pointing out at some references that were unknown formerly to authors.

References

1. Brickell E.F., Davenport D.M. On the classification of ideal secret sharing schemes. *J. Cryptology*, **4**, 1991, p.123-134.
2. Chow S., Johnson H., and Gu Y., Tamper resistant software — control flow encoding. Filed under the Patent Coöperation Treaty on August 18, 2000, under Serial No. PCT/CA00/00943.
3. Collberg C., Thomborson C., Low D., A taxonomy of obfuscating transformations, Tech. Report, N 148, Dept. of Computer Science, Univ. of Auckland, 1997.
4. Collberg C., Thomborson C., Low D., Manufacturing cheap, resilient and stealthy opaque constructs, *Symp. on Principles of Prog. Lang.*, 1998, p.184-196.
5. Collberg C., Thomborson C., Low D. Breaking abstraction and unstructuring data structures, in *IEEE Int. Conf. on Computer Languages*, 1998, p.28-38.
6. Garey M.R., Johnson D.S., Computers and Intractability, W.H Freeman and Co., San Francisco, 1979.
7. Glenn A., Larus J., Improving Data-Flow Analysis with Path Profilers. In *Proc. of the SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation*, Montreal, Canada, published as SIGPLAN Notices, **33**, N 5, 1998, pp. 72-84.
8. Horowitz S., Precise flow-insensitive May-Alias analysis is NP-hard, *TOPLAS*, 1997, **19**, N 1, p.1-6.
9. Jalali M., Hachez G., Vasserot C. FILIGRANE (Flexible IPR for Software AGent ReliANcE) A security framework for trading of mobile code in Internet, in *Autonomous Agents 2000 Workshop: Agents in Industry*, 2000.

10. Jones N.D., Muchnik S.S. Even simple programs are hard for analysis, *J. Assoc. Comput. Mach.*, 1977, **24** N 5, p.338-350.
11. Kennedy K., A Survey of Data Flow Analysis Techniques, in *Program Flow Analysis: Theory and Applications*, S.S.Muchnick and N.D.Jones (eds.). Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 5-54. '
12. Kozen D., Automata and Computability, Springer, 1997.
13. Knoop J., Ruthing O., Steffen B., Partial Dead Code Elimination, in *Proc. of the SIGPLAN '94 Conf. on Prog. Lang. Design and Implementation*, Orlando, FL, published as SIGPLAN Notices, **29**, N 6, June 1994, pp. 147-158.
14. Kotov V.E., Sabelfeld V.K., Theory of program schemata, M.:Nauka, 1991, 246 p. (in Russian)
15. Kuroda S.Y., Classes of languages and linear bounded automata, *Information and Control*, 1964, v.7, p.207-223.
16. Landi W., Undecidability of static analysis, *ACM Lett.on Prog. Lang. and Syst.*, **1**, 1992, **1**, N 4, p.323-337.
17. Mambo M., Murayama T., Okamoto E., A tentative approach to constructing tamper-resistant software, *Workshop on New Security Paradigms*, 1998, p.23-33.
18. Savitch W.J., Relationship between nondeterministic and deterministic tape complexities, *J. of Comput. and Syst. Sci.*, **4**, 1970, p.177-192.
19. SourceGuard, commercial version of HashJava, `http://www.4thpass.coml`
20. Tamper Resistant Software, `http://www/cloakware.com/technology.html`
21. Wang C., Hill J., Knight J., Davidson J., Software tamper resistance: obstructing static analysis of programs, Tech. Report, N 12, Dept. of Comp. Sci., Univ. of Virginia, 2000
22. Zakharov V. The equivalence problem for computational models: decidable and undecidable cases, *Lecture Notes in Computer Science*, **2055**, 2001, p.133-152.