A Graph Approach to Quantitative Analysis of Control-Flow Obfuscating Transformations

Hsin-Yi Tsai, Yu-Lun Huang, and David Wagner

***Abstract*—Modern obfuscation techniques are intended to discourage reverse engineering and malicious tampering of software programs. We study control-flow obfuscation, which works by modifying the control flow of the program to be obfuscated, and observe that it is difficult to evaluate the robustness of these obfuscation techniques. In this paper, we present a framework for quantitative analysis of control-flow obfuscating transformations. Our framework is based upon the control-flow graph of the program, and we show that many existing control-flow obfuscation techniques can be expressed as a sequence of basic transformations on these graphs. We also propose a new measure of the difficulty of reversing these obfuscated programs, and we show that our framework can be used to easily evaluate the space penalty due to the transformations.**

***Index Terms*—Code obfuscation, computer prime, reverse engineering, software metrics, software protection.**

I. Introduction

THE protection of software programs against illicit access is an important issue for many software companies. Since the mid 1990s, digital rights management (DRM) [1], [2] has been used to control unauthorized duplication and illegal piracy and, thus, protect the profits of publishers and owners of this software. DRM can be implemented by injecting a self-checking code into a program. This verification code is typically executed before the original program to verify the authority of the user and check that the program is being used as intended. However, attackers can still try to reverse engineer the resulting program and skip or remove the verification code if its location is not well hidden.

Recently, advanced techniques, such as software encryption [3]–[5] and software obfuscation [6]–[16] have been proposed to protect the verification logic in DRM-protected programs. In the software encryption approach, the program is encrypted and self-decrypted upon execution. Unfortunately, this can negatively affect performance. In comparison, code obfuscation works by transforming an application so that the transformed program will be functionally identical to the original one but with much greater resistance to reverse engineering. The promise of obfuscation is that obfuscated applications can run on an untrusted platform without the risk of reverse engineering, tampering, or intellectual property thefts. Code obfuscation techniques require no extra hardware and are platform independent and, as a result, they provide greater flexibility in how programs may be deployed.

Collberg *et al.* [9], [10] classified obfuscating transformations and proposed several approaches to program obfuscation. One approach is control-flow obfuscation, which tries to disguise the real control flow in a program by reordering and obfuscating the execution paths and structure of the original program. This provides a candidate way of trying to hide self-checking verification logic in the obfuscated program, thereby making that logic difficult to bypass or remove. There have been many proposals on how to perform control-flow obfuscation; however, earlier works are unable to clearly quantify the security of these control-flow obfuscation methods against reverse engineering.

This paper presents an abstract framework for formalizing and modeling many kinds of control-flow obfuscation algorithms. In this framework, we describe an obfuscation scheme as a transformation on program control-flow graphs (CFG). A control-flow obfuscation algorithm can be viewed as a function that accepts the original program's CFG as input and yields a modified CFG. By analyzing many existing control-flow obfuscating transformations, we observed that many of them can be decomposed into a sequence of basic building blocks. Thus, we identify a set of atomic operators for simple graph transformations that are guaranteed to preserve the functional behavior of the program and, hence, can be used as building blocks of a control-flow obfuscation algorithm. By composing instances of these atomic operators in sequence, we can build many kinds of control-flow obfuscating transformations. This helps to understand and classify many prior control-flow obfuscation proposals and may help in devising new candidate control-flow obfuscation methods.

We show that our framework helps to statistically analyze and evaluate control-flow obfuscating transformations. The framework only focuses on statically obfuscated source programs, and cannot be apply to dynamic analysis of reverse engineering. We also show how to evaluate the overhead on code size introduced by a control-flow obfuscation method that can be expressed within our framework. Our approach works by characterizing the space penalty of each individual atomic operator. We propose a metric that we conjecture may be related to the robustness of the obfuscated program against reverse engineering. We hope that these evaluation techniques will help to evaluate the tradeoff between the effectiveness and the overhead of different obfuscation methods.

Manuscript received May 11, 2008; revised October 06, 2008. First published February 13, 2009; current version published May 15, 2009. This work was supported in part by the iCAST Project and TWISC, sponsored by the NSC under Grants NSC96-3114-P-001-002-Y, NSC95-2218-E-001-001, NSC95-2218-E-011-015, and NSC-97-2918-I-009-005. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Jana Dittmann.

H.-Y. Tsai and Y.-L. Huang are with the Department of Electrical and Control Engineering, National Chiao-Tung University, Taiwan 300.

D. Wagner is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA.

Digital Object Identifier 10.1109/TIFS.2008.2011077

The novel contributions of this paper are as follows.

- We propose a framework with a reasonable set of atomic operators. This framework is flexible enough to adopt other types of operators as well.
- We show how to systematically formalize many existing control-flow obfuscation techniques by characterizing them as a functional composition of our atomic operators. These operators can also be used to design new control-flow obfuscating transformations.
- We propose metrics that we conjecture may be helpful in evaluating the performance and robustness of control-flow obfuscation techniques built in this framework.

This paper is organized as follows. In Section II, we give an overview of related work. Section III reviews the background of CFGs and Section IV describes the proposed atomic operators. The formalization of the control-flow obfuscating transformations is specified in Section V. Section VI proposes a metric for evaluating the robustness of transformations and we analyze the overhead of these transformations on code size in Section VII. Finally, Section VIII gives an example and the conclusion is in the last section.

II. Related Works

There are several types of obfuscating transformations [6]–[11], including layout obfuscation, data obfuscation, control-flow obfuscation, and preventive transformation. Since the control flow of a program reveals the structure of the program logic, the CFG is very helpful to a deobfuscator; consequently, to be useful for preventing reverse engineering, a program obfuscation method should obfuscate the structure of the program's control flow. Control-flow obfuscation hides or restructures the flow of execution of the program and, thus, makes reverse engineering more difficult. Techniques used for control-flow obfuscation include branch insertion, code reordering, and loop condition insertion transformation. A brief survey of these techniques will be given.

Branch insertion [9] works by inserting opaque predicates into a program to disturb and conceal the real control flow. An opaque predicate is a Boolean-valued expression whose value is known *a priori* to an obfuscator but is difficult for a deobfuscator to deduce. These opaque predicates can be categorized into three types [9]: 1) a type I opaque predicate always evaluates to false; 2) a type II predicate always evaluates to true; and 3) a type III predicate can sometimes evaluate to true and sometimes to false. In this paper, we denote these predicates by P^F, P^T, and $P^?$, respectively.

When a P^F is used, the original code is moved to the false target of the predicate to maintain the same functionality. Similarly, the original code is moved to the true target for a P^T. With a $P^?$, the original code is placed on one branch target while the other must contain some functionally equivalent copy of the original code (since we do not know in advance which branch will be taken).

The code reordering obfuscation [8] randomizes the order in which independent instructions of a program appear so that the spatial locality of the instructions will not reveal the relationship among the instructions, nor provide useful clues to the execution logic of the program. Reordering focuses on jumbling the placement of code sections in a source program.

Loop condition insertion [8] intends to increase the complexity of a loop by extending its conditions. This kind of transformations use P^F and P^T to make branch conditions more complex and further increase the difficulty of reverse engineering.

III. Control-Flow Graphs

CFGs were developed by Cota *et al.* [17], [18] as a representation of the control-flow structure of a program. We use CFGs to facilitate the formalization of obfuscating transformations. We review some basic concepts and introduce the notations representing the program's CFG.

As a high-level abstraction, a software program is composed of a sequence of code blocks. The program can be converted into a directed graph whose vertices are its code blocks. There is an edge between two code blocks if the second code block can be executed immediately after the first. In this paper, a code block is either a branch instruction or a sequence of nonbranch instructions, with notations as follows.

- Branch (B): A branch refers to an instruction that can cause execution to transfer, either conditionally or unconditionally, to some statement other than the immediately following statement. In high-level programming languages, branch instructions may be found in `for`, `while`, `do-while`, `if-else`, and `goto` statements.
- Simple block (S): A simple block is defined as an ordered sequence of statements without outgoing or incoming branch instruction inside this code block.

We will use the following notation for several special kinds of code blocks:

- entry block (C_0): the entry point of a source program;
- any block (C_A): any existing code block, or a dummy code block (C_D) that has been inserted to the program without affecting the final execution result;
- equivalent block $[\xi(C)]$: a code block that is functionally equivalent to the code block C;
- termination block (ϕ): the exit point of a source program.

The edges in a directed graph of a software program represent possible executing paths that the program might take. We introduce two kinds of edges as follows.

1) Sequential edge (e): A sequential edge $e = (C_i, C_j)$ is defined for two code blocks C_i and C_j (where $i \neq j$) if the execution of C_i is always immediately followed by executing C_j.
2) Branch edge (b): Since a branch instruction B may jump to either its true or false target, there are two code blocks that could be executed after B. The two branch edges leaving B are represented as $b^T = (B, C^{\text{True}})^T$ and $b^F = (B, C^{\text{False}})^F$, where C^{True} and C^{False} represent the true and false targets of B, respectively.

The directed graph G can be represented by the pair (V, E), where V is the vertex set and E is the edge set. V contains all of the code blocks of the parsed program, including simple blocks, branches, and a termination ϕ. E is composed of sequential edges and branch edges. Then, a software program ψ is a pair (C_0, G) of an entry point C_0 and a directed graph G. Since ϕ

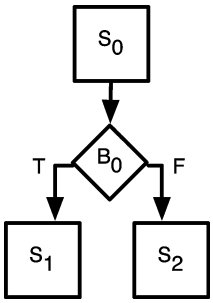

Fig. 1. Example of the formal representation of a parsed program.

is an indication of the end of the execution path, no code exists in this vertex. Hence, it is not counted into the total number of the vertex set V. Fig. 1 shows an example of a program graph ψ which contains three simple blocks and one branch, where $\psi = (S_0, G)$, $G = (V, E)$, $V = \{S_0, B_0, S_1, S_2, \phi\}$,, and $E = \{(S_0, B_0), (B_0, S_1)^T, (B_0, S_2)^F, (S_1, \phi), (S_2, \phi)\}$.

IV. Atomic Operators

A program graph is a complete representation of a source program. Obfuscating the control flow of a program can be viewed as converting one program graph to another. For graph conversion, we can use deletion, addition, and update. With deletion, a vertex or an edge is removed. As deletion always alters the functionality of the original code, we do not use deletion for program obfuscation. Addition inserts additional edges or vertices, and update means to modify the existing vertices in the graph. Although addition and update may also change the execution result, dummy or redundant codes can be used to maintain the original functionality. Therefore, control-flow obfuscation may involve two classes of operators: insertion and update. We describe these two sets of atomic operators, called "operators" and denoted by "O" hereafter.

A. Insertion

Since the code blocks are classified into only two types: 1) simple blocks and 2) branches, inserting vertices means inserting simple blocks or branches. To insert simple blocks without affecting the original functionality, we can insert dummy blocks that do nothing but resemble real code. Inserting branches can be realized by inserting opaque predicates and dummy loops.

1) Insert Dummy Simple Blocks: The insertion of dummy codes changes the control flow of a source program. Fig. 2 exhibits the operator O_D^S representing the insertion of a dummy simple block C_{D_s} in front of the target code block C_T. The graph on the right-hand side is the CFG of $O_D^S(\psi, C_T)$, which represents a result after applying O_D^S to C_T in ψ. In ψ, all edges whose successor or true/false target is C_T would be replaced. An additional sequential edge (C_{D_s}, C_T) is also inserted for the edge set E.

2) Insert Opaque Predicates: The opaque predicates can be applied by inserting branches for obfuscation to preserve the same execution result. The insertion can be accomplished by inserting the three types of opaque predicates to hide the real

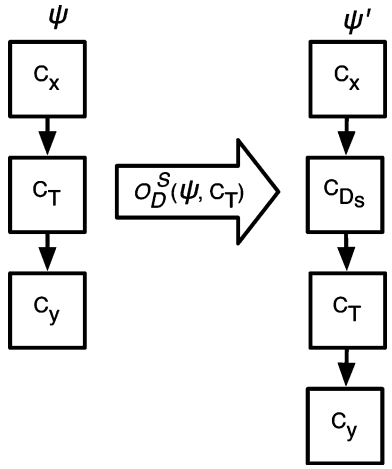

Fig. 2. Operator of inserting a dummy simple block. After insertion, E becomes $\{(C_x, C_{D_s}), (C_{D_s}, C_T), (C_T, C_y)\}$.

control flow of a source program. O_{Op}^F, O_{Op}^T, and $O_{Op}^?$ represent the three types of insertion: type I (false), II (true), and III, respectively.

type I) O_{Op}^F: As P^F is inserted in front of the target block C_T, C_T should be moved to the false target of P^F to maintain the same functionality [see Fig. 3(a)]. Since the execution result of P^F is always false, any code block C_A may be specified as the true target of P^F. C_A can be an existing or a dummy code block by applying the operator O_D.

type II) O_{Op}^T: The procedure for inserting P^T is similar to that for P^F [see Fig. 3(b)]. Since P^T always evaluates to true, C_T is placed as the true target of P^T. C_A, any code block, can be its never-reached false target.

type III) $O_{Op}^?$: Fig. 3(c) shows the actions of $O_{Op}^?$. To ensure the same functionality, the equivalence of C_T is placed on one of the targets of $P^?$.

3) Insert Dummy Loops: A loop can be achieved by combining simple blocks and branches. The operator O_D^L inserts an extra loop in front of the target block C_T as shown in Fig. 4. If C_T is the successor of a sequential edge or the true/false target of a branch edge, it will be replaced by a dummy branch C_{D_b}. Then, a new sequential edge (C_{D_s}, C_{D_b}) and two additional branch edges $(C_{D_b}, C_{D_s})^T$, $(C_{D_b}$ and $C_T)^F$ are inserted into the edge set, where C_{D_s} is a dummy simple block. In this way, a loop composed of C_{D_s} and C_{D_b} is constructed.

B. Update

We consider splitting, reordering, and replacing operators for modifying a vertex.

1) Split Code Blocks: Splitting a code block into pieces can increase the number of vertices in the CFG and increase its complexity. Combining the splitting operator with other operators helps implement more complex obfuscating transformations. In the following actions, the operators of splitting simple blocks and splitting branches are explained.

- $O_S^{S,n}$: The operator aims at splitting a simple block into n pieces. In Fig. 5, $O_S^{S,n}(\psi, C_T)$ splits C_T into n pieces, where n is limited to the instruction counts of C_T.

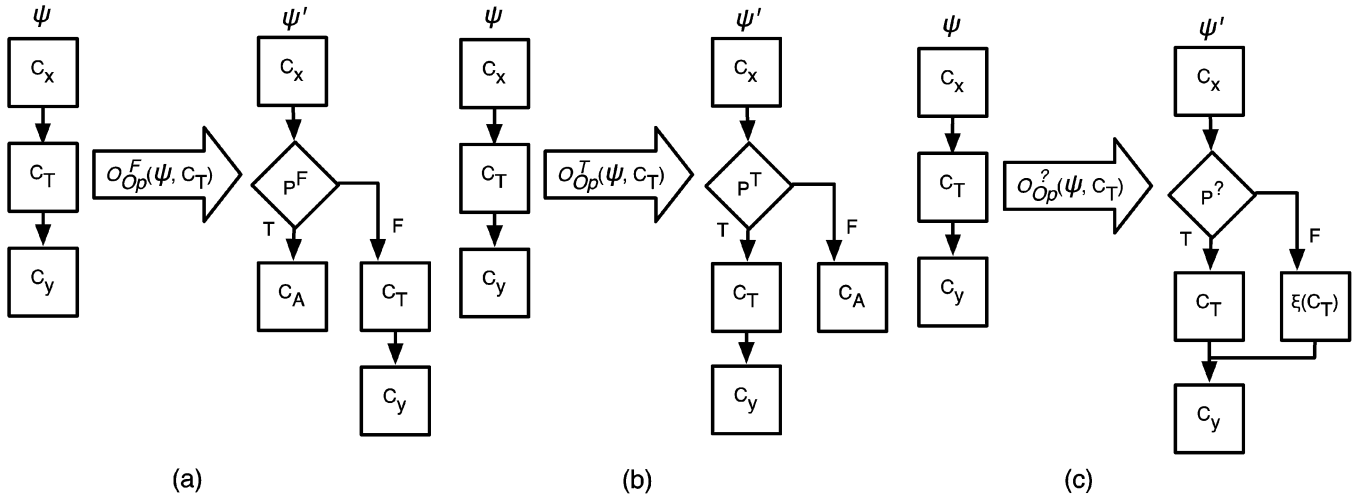

Fig. 3. Operators of inserting opaque predicates: (a) Type I, (b) Type II, and (c) Type III.

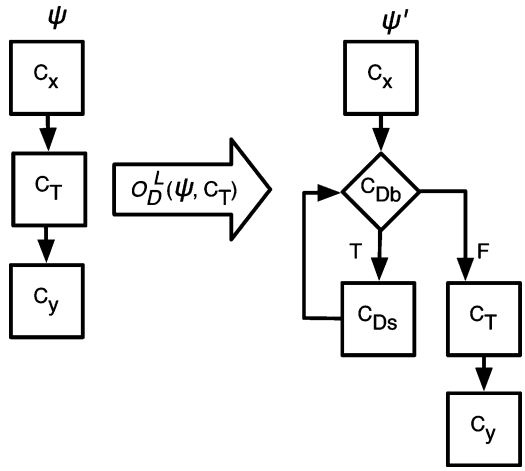

Fig. 4. Operator of inserting dummy loops. After insertion $E = \left\{(C_x, C_{D_b}), (C_{D_b}, C_{D_s})^T, (C_{D_b}, C_T)^F, (C_{D_s}, C_{D_b}), (C_T, C_y)\right\}$.

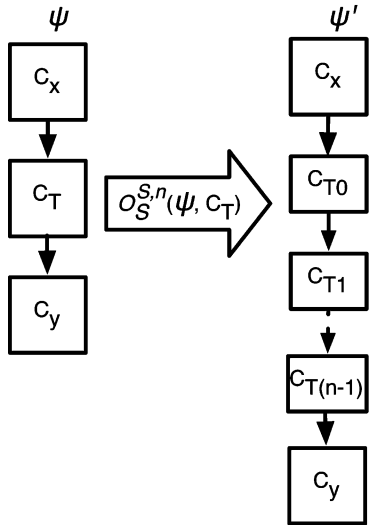

Fig. 5. Operator of splitting a simple block. After splitting, $E = \{(C_x, C_{T0}), (C_{T0}, C_{T1}), \ldots, (C_{T(n-2)}, C_{T(n-1)}), (C_{T(n-1)}, C_y)\}$.

- $O_S^{B,n}$: The operator splits a target branch into smaller pieces. Similarly, the parameter n is limited to the numbers of conditions in C_T. In Fig. 6, C_T can be expressed as $cond_0$ AND $(($$cond_1$ AND $cond_2)$ OR $cond_3)$. Since there are four conditions in C_T, n is limited to 4. These conditions are first converted to postfix orders $cond_0$ $cond_1$ $cond_2$ AND $cond_3$ OR AND. After splitting C_T, the original CFG is then converted to the CFG on the right-hand side, where C_{T0}, C_{T1}, C_{T2}, and C_{T3} represent $cond_2$, $cond_1$, $cond_3$, and $cond_0$, respectively.

2) Reorder Code Blocks: Randomizing the placement of instructions helps to hide the original execution logics from being reversely engineered. The reordering operator O_R then becomes one of the operators in obfuscating programs (see Fig. 7). Before applying O_R, the execution dependency between C_T and its immediate successor C_{T+1} should be checked. If dependency exists, then O_R may result in an incorrect execution.

3) Replace With Equivalent Codes: Equivalent codes are those that have the same execution result as the origins, while their implementations are different. The equivalent codes conduce to confuse reverse engineers. The operator $O_E(\psi, C_T)$ replaces C_T in ψ with its equivalent code $\xi(C_T)$.

V. Formalization of Obfuscating Transformations

A control-flow obfuscating transformation $\mathcal{T}$ can be decomposed into a sequence of operators. Different sequences of operators lead to different transformations. Even with the same sequence, specifying different target blocks to these operators may obtain different results. We represent a transformation $\mathcal{T} = \langle f_1, f_2, \ldots, f_m \rangle$ as the composition of m operators $f_1, \ldots, f_m$, where $f_x \in \left\{O_{Op}^F(\cdot, C_a), O_{Op}^T(\cdot, C_b), O_{Op}^?(\cdot, C_c), O_E(\cdot, C_d), O_S^{S,n}(\cdot, C_e), O_S^{B,n}(\cdot, C_f), O_D^L(\cdot, C_g), O_D^S(\cdot, C_h), O_R(\cdot, C_i)\right\}$, for $x = 1, \ldots, m$. Note that $\langle\rangle$ stands for an ordered set of functional composition, where $\langle f_1, f_2, \ldots, f_m \rangle$ represents the function g defined by $g(x) = f_m(\ldots f_2((f_1(x))\ldots)$. $C_a, C_b, \ldots, C_k$ represents arbitrary code blocks from the source program.

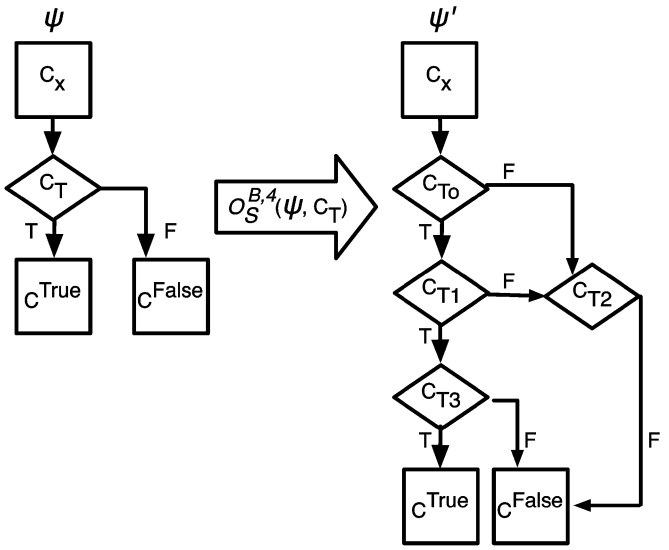

Fig. 6. Operator of splitting a branch. The example splits C_T into four pieces. After splitting, $E = \Big\{(C_x, C_{T0}), (C_{T0}, C_{T1})^T, (C_{T0}, C_{T2})^F, (C_{T1}, C_{T3})^T, (C_{T1}, C_{T2})^F, (C_{T2}, C_{T3})^T, (C_{T2}, C^{\text{False}})^F, (C_{T3}, C^{\text{True}})^T, (C_{T3}, C^{\text{False}})^F\Big\}$.

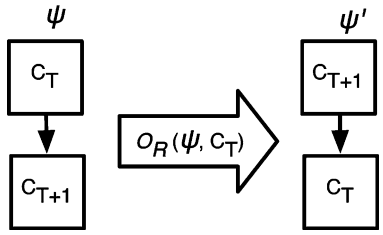

Fig. 7. Operator of reordering code blocks. After reordering, $\{(C_T, C_{T+1})\}$ is replaced with $\{(C_{T+1}, C_T)\}$.

This formal model can be used to describe many existing control-flow transformations [6], [9], [12]–[16], according to their algorithms. Decomposing these transformations into a sequence of operators also enables further analysis. Table I classifies 17 existing transformations according to whether they can be represented as a functional composition of our operators. As the table shows, ten transformations can be decomposed into a sequence of the proposed operators, but six of them cannot. In this section, we detail the decomposition, justify each entry in the table, and interpret these results.

A. Basic Block Fission Obfuscation [6]

This obfuscation tries to subvert the structures of programs so that decompiling the transformed programs would be unsuccessful. This transformation splits the chosen code blocks into more pieces, and inserts opaque predicates and `goto` instructions into these pieces. In the example presented in [6], to protect against the decompilation attack, a few more blocks were generated and inserted after splitting the chosen code blocks. Then, a type I opaque predicate was inserted to make sure of the unreachability of the newly inserted code blocks and, thus, functionality of the original program was preserved.

TABLE I
FEASIBILITY OF DECOMPOSITION

Control Flow Obfuscating Transformations	Decomposable?
Basic Block Fission Obfuscation [6]	Y
Intersecting Loop Obfuscation [6]	Y
Replacing goto Obfuscation [6]	Y
Branch Insertion Transformation [9]	Y
Loop Condition Extension Transformation [9]	Y
Language-Breaking Transformation [9]	Y
Parallelize Code [9]	N
Add Redundant Operands [9]	Y
Aggregation Transformations [9]	N
Ordering Transformations [9]	Y
Remove Library Calls and Programming Idioms [9]	C
Table interpretation [9]	N
Degeneration of control flow [12]	Y
Obfuscation Scheme Using Random Numbers [13]	Y
Obfuscating C++ Programs via Flattening [14]	N
Control Flow Based Obfuscation [15]	N
Binary Obfuscation Using Signals [16]	N

Y: can be expressed N: cannot be expressed C: conditional

In this case, according to the type of the chosen code blocks, we can apply $O_S^{S,n}$ to a simple block or $O_S^{B,n}$ to a branch. Moreover, O_D^S and O_D^L can be used to insert dummy code blocks. Type II opaque predicates are used to perform the functionality of `goto` instructions while any one of three O_{Op} operators can be inserted as opaque predicates to realize the basic block fission obfuscation. Thus, this transformation can be expressed as $\mathcal{T} = \langle f_1, f_2, f_3, f_4\rangle$, where $f_1 \in \left\{O_S^{S,n}, O_S^{B,n}\right\}$, $f_2 \in \left\{O_D^S, O_D^L\right\}$, $f_3 = O_{Op}^T$, and $f_4 \in \left\{O_{Op}^F, O_{Op}^T, O_{Op}^?\right\}$.

B. Intersecting Loop Obfuscation [6]

This obfuscation inserts two intersected loops to a source program to make control flows unrecognizable for decompilers. Also, a type I opaque predicate is inserted to skip the newly inserted intersecting loops and to avoid any influence upon the original execution. Since a loop consists of a simple block and a branch, we use two simple blocks and two opaque predicates to create the two intersected loops. To preserve the same execution, the newly inserted loops are followed by a type I opaque predicate. Hence, this transformation can be expressed as $\mathcal{T} = \left\langle O_D^S, O_D^S, O_{Op}, O_{Op}, O_{Op}^F\right\rangle$, where $O_{Op} \in \left\{O_{Op}^F, O_{Op}^T, O_{Op}^?\right\}$.

C. Replacing `goto` Obfuscation [6]

This obfuscation replaces `goto` instructions with conditional branch instructions that do not influence the original control flow. This can be realized by replacing the go to instructions with their equivalent codes. The transformation can be represented as $\mathcal{T} = \langle f_1, f_2, \ldots, f_m\rangle$, where $f_x = O_E$ for $x = 1, \ldots, m$.

D. Branch Insertion Transformation [9]

This transformations is designed based on one of the three opaque predicate insertion operators O_{Op}^F, O_{Op}^T, and $O_{Op}^?$. It can be expressed as $\mathcal{T} = \left\langle O_S^2, O_{Op}, [O_D]\right\rangle$. The target block is first split into two pieces by $O_S^2 \in \left\{O_S^{S,2}, O_S^{B,2}\right\}$. The second step is to apply O_{Op} to the split pieces, where

$O_{Op} \in \{O_{Op}^F, O_{Op}^T, O_{Op}^?\}$. Finally, the insertion of dummy codes $O_D \in \{O_D^S, O_D^L\}$ is optional in this transformation.

E. Loop Condition Extension Transformation [9]

A loop can be obfuscated by complicating the loop condition. The idea is to extend the loop condition using opaque predicates that do not affect the iterations when the loop is executed. The targets of opaque predicates are the branch blocks forming the loop condition. These opaque predicates are inserted immediately in front of the branch blocks. Optionally, a dummy code block can also be placed in its never-reached target. The formal representation of this transformation can be defined as $\mathcal{T} = \langle O_{Op}, [O_D] \rangle$, where $O_{Op} \in \{O_{Op}^F, O_{Op}^T, O_{Op}^?\}$ and $O_D \in \{O_D^S, O_D^L\}$ (optional).

F. Language-Breaking Transformation [9]

This transformation converts a reducible flow graph to a nonreducible one by turning a structured loop into a loop with multiple headers. For obscurity, the loop body is split into two pieces. A type I or type II opaque predicate is inserted in front of the original loop to make a never-executed jump into the second split piece. Since it is a never-executed jump, the second split piece is placed on the never-executed target of the inserted opaque predicate. The expression in terms of the operators is defined as $\mathcal{T} = \langle O_S^2, O_{Op}, [O_D] \rangle$, where O_S^2 is the operator to split a code block into two halves $O_{Op} \in \{O_{Op}^F, O_{Op}^T\}$, and $O_D \in \{O_D^S, O_D^L\}$ is optional.

G. Parallelize Code [9]

A reverse engineer may find a parallel program more difficult to understand than a sequential one. Thus, parallelization may lead to higher potency. To increase parallelism for obscuring the control flow of a program, we can either create dummy processes or split a code block into multiple data-independent blocks executing in parallel. Since parallel execution cannot be expressed using a simple graph representation, it fails to decompose this transformation in our framework.

H. Add Redundant Operands [9]

Algebraic laws can be used to add redundant operands to arithmetic expressions. The logic of the original expression is modified, and the operation becomes more complex. The transformation is formalized by $\mathcal{T} = \langle f_1, f_2, \ldots, f_m \rangle$, where $f_x = O_E$ for $x = 1, \ldots, m$. Only the method "add redundant codes" can be used as the technique of the creation of equivalent codes for the operator O_E.

I. Aggregation Transformations [9]

This transformation falls into two categories. One is to break up codes where programmers aggregated them into a method and scatter the codes over the program. The other is to aggregate the codes which seem to not belong together into one method. Since operators are mainly applied to code blocks, this transformation with the basis of methods cannot be represented by using our operators.

J. Ordering Transformations [9]

To eliminate useful spatial clues to understand the execution logics of a program, ordering obfuscation was proposed to randomize the placement of any code block in a source program. The operator $O_R(\psi, C_T)$ is used to express the ordering transformations in the form $\mathcal{T} = \langle f_1, f_2, \ldots, f_m \rangle$ where $f_x = O_R$ for $x = 1, \ldots, m$. Note that O_R exchanges the two target blocks if no dependency exists between them.

K. Remove Library Calls and Programming Idioms [9]

It is known that the standard JAVA library calls may provide useful clues to reverse-engineers. To impede this problem from being exacerbated, an obfuscator may provide its own versions of the standard libraries. If this transformation is designated to apply to code blocks, instead of programs, the target block can be replaced with its equivalent codes and expressed as follows. $\mathcal{T} = \langle f_1, f_2, \ldots, f_m \rangle$, where $f_x = O_E$ for $x = 1, \ldots, m$.

L. Table Interpretation [9]

This transformation converts a code block into a different virtual machine code which is then executed by a virtual machine interpreter within the obfuscated program. Since we do not talk about interpreters in this paper, it fails to formalize this transformation with the proposed operators.

M. Degeneration of Control Flow [12]

This transformation converts high-level control structures into equivalent *if-then-goto* constructs. Then, `goto` statements are modified such that the target addresses of the `goto` statements are computed at runtime. In the first step, the expected construct can be developed according to the proposed CFG. Since the transformation replaces control flow with computed-`goto` statements, equivalence techniques can be used to generate the target blocks of the `goto` statements. Subsequently, O_E can be applied to branches of the construct to dynamically determine the target address of the `goto` instructions. Thus, the transformation can be expressed as $\mathcal{T} = \langle f_1, f_2, \ldots, f_m \rangle$, where $f_x = O_E$ for $x = 1, \ldots, m$.

N. Obfuscation Scheme Using Random Numbers [13]

In this transformation, a dispatcher uses a random number (RN) to determine its target method while a method point (MP) is used to check whether the selected target method should be executed or not. If RN $\neq$ MP, the selected method is not executed. The transformation regenerates a random number to select another method until RN matches MP.

The concept of using a dispatcher and a random number can be accomplished by the obscurity and randomness of type III opaque predicates. Here, a type III opaque predicate is inserted in front of each method designated as the true target of the predicate. If the predicate evaluates true, its corresponding method is reached; otherwise, the execution jumps to another predicate with the same functionality as the former. Since MP is used to determine the accurate execution path, we insert other type III opaque predicates for each method, where the newly inserted

predicates play the same role as MP. Hence, the transformation can be expressed in the form $\mathcal{T} = \langle f_1, f_2, \ldots, f_m \rangle$, where $f_x = O^?_{Op}$ for $x = 1, \ldots, m$.

O. Obfuscating C++ Programs Via Flattening [14]

The transformation is to first break up the function body into several smaller blocks and then make the blocks in the same nesting level. Besides, a dispatcher determines which equal-leveled blocks are to be executed. Although we can adopt the same way for the implementation of the dispatcher, we cannot carry out the main idea of this transformation that the split blocks are in the same nesting level. Therefore, it is unable to express the transformation with the operators.

P. Control-Flow-Based Obfuscation [15]

Two processes, P and M, are used in this transformation. The P-process performs the main functionality and acts as the original program. The M-process handles and saves the control-flow information extracted from the original program. P-process queries the M-process for the correct addresses whenever the P-process reaches a point with missing control-flow information. Since additional information is needed to achieve this transformation, we fail to decompose it.

Q. Binary Obfuscation Using Signals [16]

This transformation replaces an unconditional jump with code, attempting to access an illegal memory location that raises a signal. The signal-handling routine determines the target address of the original unconditional jump and takes over the control flow of the program. Since we do not refer to any signals and signal-handling routines, this transformation cannot be expressed with our operators.

VI. Evaluation Metrics

Reverse engineers generally follow the following process [19] to reverse-engineer a program:

- identify the component that will be reverse engineered;
- observe the execution flow, read manuals, and disassemble the code.

The difficulty of reverse engineering an obfuscated program depends on the relationship between the original and transformed program. The exact amount of effort required is difficult to quantify, because it depends upon the experience and skill level of the deobfuscator: it may take some people significantly longer than others to reverse-engineer the same program.

We propose a measure that tries to eliminate factors varying from person to person. Our measure does not compare the difficulty of reverse engineering the same program between different reverse engineers; rather, it is intended to estimate the difficulty of reversing different obfuscated programs, if we hold constant the person who is performing the reverse engineering. Our approach is to define a distance metric that reflects the degree of difference between the original program and the obfuscated program. In this paper, we use distance and potency metrics to evaluate the robustness of obfuscated programs. We recognize they serve as merely heuristic, general indicators of security. However, these metrics can still be the first step toward evaluation of robustness.

We do not claim that a large value of our metric implies that the obfuscation will necessarily be secure against reverse engineering; we expect that large values of this metric are necessary but not sufficient for security. Our metric is only intended to reflect the difficulty of reverse engineering through static analysis—it does not reflect information that might be gained by running the program and observing its execution, or by performing some other kind of dynamic analysis. Nonetheless, we conjecture that this metric may be helpful in comparing different approaches to obfuscation.

A. Distance Metric

Bunke [20] proposed a distance metric based on the maximal common subgraph (MCS). The distance between two graphs is given in terms of the number of nodes of their MCS

$$d(G_1, G_2) = 1 - \frac{|\mathrm{mcs}(G_1, G_2)|}{\max(|G_1|, |G_2|)}.$$

Here, $|G|$ is the number of nodes of the graph G, and $\mathrm{mcs}(G_1, G_2)$ represents the MCS of G_1 and G_2. This distance metric could be used to measure the robustness of a control-flow obfuscation method by letting G_1 denote the CFG of the original program and G_2 be the CFG of the obfuscated program.

Wallis *et al.* [21] proposed another distance metric

$$d(G_1, G_2) = 1 - \frac{|\mathrm{mcs}(G_1, G_2)|}{|G_1| + |G_2| - |\mathrm{mcs}(G_1, G_2)|}.$$

We refer to this as the graph union method, since $|G_1| + |G_2| - |\mathrm{mcs}(G_1, G_2)|$ is loosely related to the size of the graph union. It is exactly the size of the union if G_1 and G_2 have only one common subgraph. These two metrics only consider the size of the MCS, and do not reflect any changes in other common subgraphs. As a result, they may fail to accurately measure the robustness of some control-flow obfuscating transformations.

Our distance metric differs from those of earlier works in that we take all common subgraphs into account, not merely the MCS. We also count the number of edges in these common subgraphs to reflect possible execution paths. Our metric measure of graph edge (MGE) quantifies the distance between two graphs G_1 and G_2

$$d(G_1, G_2) = 1 - \sum_i \frac{2\,|\mathrm{edge}(CS_i(G_1, G_2))|}{|\mathrm{edge}(G_1)| + |\mathrm{edge}(G_2)|} \qquad (1)$$

where $CS_i(G_1, G_2)$ refers to the ith common subgraph of G_1 and G_2, $\mathrm{edge}(G)$ is the set of edges within graph G, and $|\mathrm{edge}(G)|$ is the number of edges within G. The minimum value of $d(G_1, G_2)$ is "0" if the two graphs are exactly the same. The maximum value of $d(G_1, G_2)$ is "1" if no common subgraph exists between G_1 and G_2.

Assume that we know which vertices in G_2 correspond to which vertices in G_1, then the common subgraphs can be uniquely identified and the distance metric is well defined. Fig. 8 gives an example of graphs G_1 and G_2, both having eight nodes and seven edges. There are three common subgraphs with 0, 1, and 3 edges. According to (1), we obtain $d(G_1, G_2) = 3/7$.

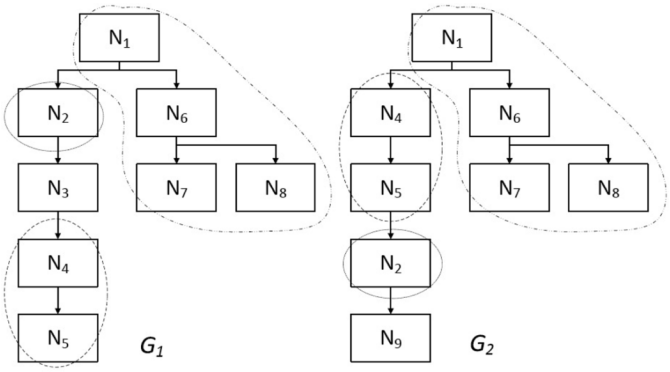

Fig. 8. Two graphs G_1 and G_2. Three common subgraphs of G_1 and G_2 are circled.

B. Potency Metric

To measure the complexity and overhead of obfuscated programs, Collberg *et al.* [9] proposed several metrics for evaluating an obfuscating transformation, including cost, resilience, and potency. The cost metric measures the additional runtime resources required to execute an obfuscated program. The resilience metric is intended to measure how well an obfuscating transformation holds up against attacks from an automatic deobfuscator. The potency metric is supposed to be related to the degree to which an obfuscating transformation confuses a human trying to understand the obfuscated program. Of these three metrics, only potency is intended to measure the difficulty for a reverse engineer to compromise and deduce an obfuscated program. The potency is defined as

$$p(PG, PG') = \frac{\text{comp}(PG')}{\text{comp}(PG)} - 1. \tag{2}$$

Here, $\text{comp}(PG)$ and $\text{comp}(PG')$ denote the complexity of the original program PG and the obfuscated program PG'.

Many methods have been proposed to evaluate the complexity of software programs, including measure relative logical complexity (RLC), absolute logical complexity (ALC), and N-Scope [22]. The complexity of a program is measured by the number of edges, branches, and nodes in its CFG. RLC uses the ratio of the numbers of branches and nodes to represent the complexity. ALC counts branches only. These two measures may not fully reflect the true complexity of the program: two different obfuscated programs with the same value of these metrics may not seem equally complex to a human trying to reverse-engineer the program. Consequently, computing the potency metric using RLC or ALC may not accurately characterize the robustness of obfuscation techniques.

The value of the N-Scope complexity metric is determined by the nesting levels of all branches in a program. The N-Scope complexity metric (ns) is given by

$$ns(\psi) = \frac{\sum_{B_i \in B} |R(\psi, B_i)|}{\sum_{B_i \in B} |R(\psi, B_i)| + |NC_\psi|} \tag{3}$$

where B is the set of branch blocks in ψ, $|NC_\psi|$ is the node count (NC) in ψ, and $|R(\psi, B_i)|$ represents the nesting level that the branch B_i contributes. It denotes the number of nodes in the loop led by B_i or are on the paths branching out at B_i until the paths converge. The N-Scope value derived from an operator is determined by R and NC. The value can be different per the operators. Some operators (e.g., O_R and O_E) do not affect R and NC; some (e.g., $O_S^{S,n}$ and O_D^S) may contribute nothing to R but increase NC; some others (e.g., O_{Op}) always change R and NC.

C. DP Vector

In evaluating the difficulties that reverse engineers may encounter, Collberg *et al.* proposed the potency [(2)] as an estimate of the degree. However, the potency metric, if computing using the N-Scope, fails to detect all changes to execution paths and may not accurately measure the robustness of some obfuscating transformations. One way to remedy this kind of shortcoming is to devise a special distance measure for quantifying the difference between two programs. Our distance metric addresses this drawback and reflects changes that do not change the depths of loops. Therefore, we suggest using potency and our distance measure to evaluate the robustness, namely

$$DP(\psi, \mathcal{T}) = (d(\psi, \mathcal{T}(\psi)), p(\psi, \mathcal{T}(\psi))) \tag{4}$$

where ψ represents the CFG of the original program, $\mathcal{T}$ is the obfuscating transformation, and $\mathcal{T}(\psi)$ is the obfuscated CFG. Here, $p(\psi, \mathcal{T}(\psi)))$ denotes the potency computed using the N-Scope, and $d(\psi, \mathcal{T}(\psi))$ is computed by using the MGE defined in (1). We expect that larger distance and potency may be correlated to better robustness against reverse engineering.

VII. Space Penalty

Control-flow obfuscation uses techniques, such as creating buggy loops and inserting dummy codes, to disturb the real execution path. After obfuscating transformations, a source program can forbid malicious tampering and reverse engineering. However, it suffers from space penalty. The more transformations that are applied to the program, the more code size overheads suffer. Thus, estimation of the space penalty is important for assurance whether the increment of code sizes due to the designated transformations is tolerable. Through the proposed formal representation, estimation of space penalty can be efficiently determined in advance so that users can decide whether to apply more transformations or not. In this section, we analyze overheads on code sizes resulting from each operator.

Assuming that an original parsed program ψ has n code blocks where the size of the ith block is denoted as z_i, $\forall\ i \in [1, n]$, and the total code size of ψ is $\sum_{i=1}^{n} z_i$. After obfuscating transformations, x simple blocks and y branches are inserted into ψ where the size of the ith simple block and the jth branch are, respectively, indicated as s_i and $b_j, \forall\, i \in [1, x]$, and $\forall\ j \in [1, y]$. Hence, the total code size of the obfuscated program becomes $\sum_{i=1}^{n} z_i + \sum_{i=1}^{x} s_i + \sum_{i=1}^{y} b_i$ and the space penalty is $\sum_{i=1}^{x} s_i + \sum_{i=1}^{y} b_i$.

For simplicity of analysis, the summation of the sizes of all inserted blocks is replaced with the product of the average size and the number of blocks. Since the gap between the average size of simple blocks and that of branches is too large to be ignored, they should be individually denoted by $\bar{S}$ and $\bar{B}$. The space penalty becomes $x \cdot \bar{S} + y \cdot \bar{B}$. We describe the space penalty

TABLE II
SPACE PENALTY OF EACH ATOMIC OPERATOR

Atomic Operators	Space Penalty
Insert type I/II opaque predicates, O_{Op}^F/O_{Op}^T	$\bar{B}$
Insert type III opaque predicates, $O_{Op}^?$	$\bar{S}+\bar{B}$ or $2\cdot\bar{B}$
Split code blocks, $O_S^{S,n}/O_S^{B,n}$	0
Reorder code blocks, O_R	0
Replace with equivalent codes, O_E	0
Insert dummy simple blocks, O_D^S	$\bar{S}$
Insert dummy loops, O_D^L	$\bar{S}+\bar{B}$

with respect to each proposed operator, and Table II makes the arrangement.

- O_{Op}^F/O_{Op}^T introduces an extra predicate which results in a space penalty of $\bar{B}$.
- $O_{Op}^?$ inserts a new predicate and an equivalent block that contributes a space penalty of either $\bar{S}+\bar{B}$ or $2\cdot\bar{B}$, depending on the type of C_T. If C_T is a simple block, then the space penalty is $\bar{S}+\bar{B}$; otherwise $2\cdot\bar{B}$.
- $O_S^{S,n}/O_S^{B,n}$ splits C_T into smaller pieces. The space penalty is "0", but the number of nodes increases.
- O_R adds nothing and has "0" space penalty.
- O_E replaces C_T with its equivalence $\xi(C_T)$. Since it is a replacement, there is no space penalty.
- O_D^S inserts an extra simple block and gets an space penalty $\bar{S}$.
- O_D^L inserts a dummy loop containing a branch and a simple block. Thus, the space penalty is $\bar{S}+\bar{B}$.

VIII. EXAMPLE: PRIME NUMBER GENERATOR

We show how the proposed formalization method can be applied to a program, and give the evaluation after obfuscation.

A. Graph Conversion

Program I, generating prime numbers smaller than a, is used as an example to demonstrate how the proposed method works.

```
/* Program I. Prime number generator */
int k(int b) {
    int i;
 for (i = 2; i <= b/2; i++);
  if (b%i == 0) return 0;
   return 1;
}
int main() {
    int a, b, sum;
    printf("insert a number \n");
    scanf("%d", &a);
    for(sum = 0, b = 2; b <= a; b++) {
        if(k(b)) printf("%3d", b);
        sum += b;
    } return 0;
}
```

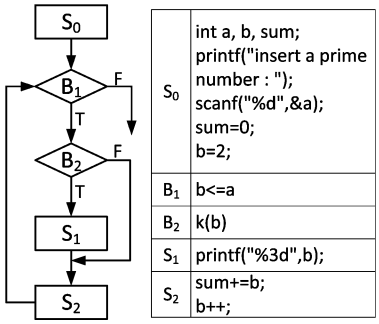

Fig. 9. Parsed CFG of Program I $\psi = (C_0,(V,E))$, where $C_0 = S_0$, $V = \{S_0,S_1,S_2,B_1,B_2,\phi\}$, and $E = \left\{(S_0,B_1),(B_1,B_2)^T,(B_1,\phi)^F,(B_2,S_1)^T,(B_2,S_2)^F,(S_1,S_2),(S_2,B_1)\right\}$.

We parse Program I and derive its CFG $\psi=(C_0,(V,E))$, as illustrated in Fig. 9.

B. Obfuscation

We apply two control-flow obfuscating transformations: 1) the basic block fission obfuscation [6] and 2) the branch insertion transformation [9] in the example

$$\mathcal{T}_1 = \left\langle O_S^{S,2}(\cdot,S_0), O_D^S(\cdot,B_1), O_{Op}^T(\cdot,B_1), O_{Op}^F(\cdot,S_{01})\right\rangle$$
$$\mathcal{T}_2 = \left\langle O_S^{S,2}(\cdot,S_0), O_{Op}^F(\cdot,S_{01}), O_E(\cdot,S_{01}), O_D^S(\cdot,B_1)\right\rangle.$$

1) Apply the Specified Basic Block Fission Obfuscation $\mathcal{T}_1$:

- Running $O_S^{S,2}(\cdot,S_0)$

  $C_0 \leftarrow S_{00}$,
  $V \leftarrow (V-\{S_0\})\cup\{S_{00},S_{01}\}$,
  $E \leftarrow (E-\{(S_0,B_1)\})\cup\{(S_{00},S_{01}),(S_{01},B_1)\}$.

  In this example, S_{00} is

  `int a, b, sum; printf("insert a number \n")`

  and S_{01} is

  `scanf("%d", &a); sum = 0; b = 2.`

  Since splitting S_0 does not contribute to the range (R) but increases the node count (NC), ns becomes 5/11.
- Running $O_D^S(\cdot,B_1)$:

  $V \leftarrow V\cup\{C_{D_{s1}}\}$,
  $E \leftarrow (E-\{(S_{01},B_1)\})\cup\{(S_{01},C_{D_{s1}}),(C_{D_{s1}},B_1)\}$.

  We choose `sum = a + b%4` as $C_{D_{s1}}$. Inserting $C_{D_{s1}}$ only increases NC, so ns becomes 5/12.
- Running $O_{Op}^T(\cdot,B_1)$

$V \leftarrow V\cup\{P_1^T\}$,
$E \leftarrow (E-\{(S_2,B_1),(C_{D_{s1}},B_1)\})$

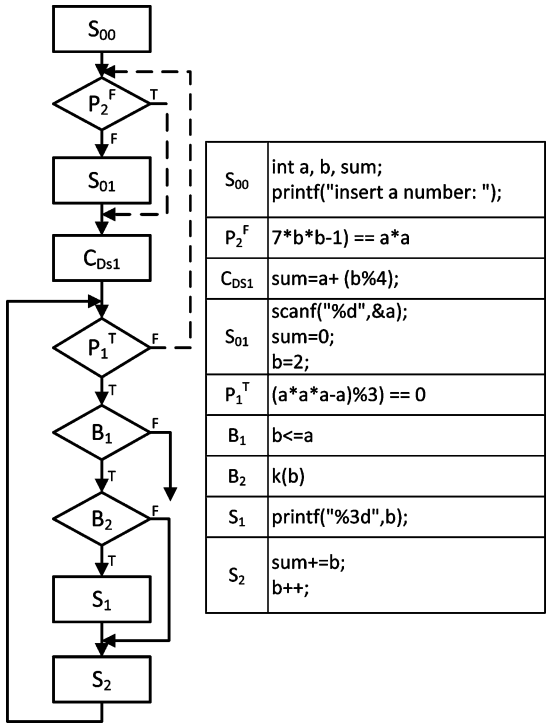

Fig. 10. Program II: the obfuscated result of Program I after applying $\mathcal{T}_1$.

$$\cup \left\{ \left(S_2, P_1^T\right), \left(C_{D_{s1}}, P_1^T\right), \left(P_1^T, B_1\right)^T, \left(P_1^T, S_{01}\right)^F \right\}.$$

We choose $(\mathtt{a}^3 - \mathtt{a})\%3 == 0$ for P_1^T, which only works with an integer a. Since R and NC increase, ns raises from 5/12 to 9/17.

- Running $O_{Op}^F(\cdot, S_{01})$

$$V \leftarrow V \cup \left\{P_2^F\right\},$$
$$E \leftarrow \left(E - \left\{(S_{00}, S_{01}), \left(P_1^T, S_{01}\right)^F\right\}\right)$$
$$\cup \left\{ \left(S_{00}, P_2^F\right), \left(P_2^F, C_{D_{s1}}\right)^T, \left(P_2^F, S_{01}\right)^F, \left(P_1^T, P_2^F\right)^F \right\}.$$

We choose $7\mathtt{b}^2 - 1 = \mathtt{a}^2$ as P_2^F.

After applying $\mathcal{T}_1$ and obtaining ψ_1, the obfuscated program (Program II) is generated according to ψ_1 (see Fig. 10).

2) Apply the Specified Branch Insertion Transformation $\mathcal{T}_2$:

- Running $O_S^{S,2}(\cdot, S_0)$

$$C_0 \leftarrow S_{00},$$
$$V \leftarrow (V - \{S_0\}) \cup \{S_{00}, S_{01}\},$$
$$E \leftarrow (E - \{(S_0, B1)\}) \cup \{(S_{00}, S_{01}), (S_{01}, B1)\}.$$

Here, we use the same S_{00} and S_{01}.

- Running $O_{Op}^F(\cdot, S_{01})$

$$V \leftarrow V \cup \left\{P_1^F\right\},$$
$$E \leftarrow (E - \{(S_{00}, S_{01})\})$$
$$\cup \left\{ \left(S_{00}, P_1^F\right), \left(P_1^F, B_1\right)^T, \left(P_1^F, S_{01}\right)^F \right\}.$$

We choose $7\mathtt{b}^2 - 1 = \mathtt{a}^2$ as P_1^F. $O_{Op}^F(\cdot, S_{01})$ inserts one more code block, thus makng NC increase by one. R is increased by one as well. Now, ns decreases to 6/13.

- Running $O_E(\cdot, S_{01})$:

$$V \leftarrow (V - S_{01}) \cup \{\xi(S_{01})\},$$
$$E \leftarrow \left(E - \left\{(S_{01}, B1), \left(P_1^F, S_{01}\right)^F\right\}\right)$$
$$\cup \left\{(\xi(S_{01}), B1), \left(P_1^F, \xi(S_{01})\right)^F\right\}.$$

Here, $\xi(S_{01})$ is generated by inserting dummy instructions. Since O_E contributes nothing to R and NC, ns remains unchanged.

- Running $O_D^S(\cdot, B_1)$:

$$V \leftarrow V \cup \{C_{D_{s1}}\},$$
$$E \leftarrow \left(E - \left\{(\xi(S_{01}), B1), (S_2, B_1), \left(P_1^F, B_1\right)^T\right\}\right)$$
$$\cup \left\{(\xi(S_{01}), C_{D_{s1}}), (S_2, C_{D_{s1}}), \left(P_1^F, C_{D_{s1}}\right)^T, (C_{D_{s1}}, B_1)\right\}.$$

We choose $\mathtt{sum} = \mathtt{a} + \mathtt{b}\%4$ as $C_{D_{s1}}$ and insert the block into the loop. Now, R and NC are both increased and ns changes from 6/13 to 7/15.

Now, we regenerate the obfuscated program (Program III) according to ψ_2 (see Fig. 11).

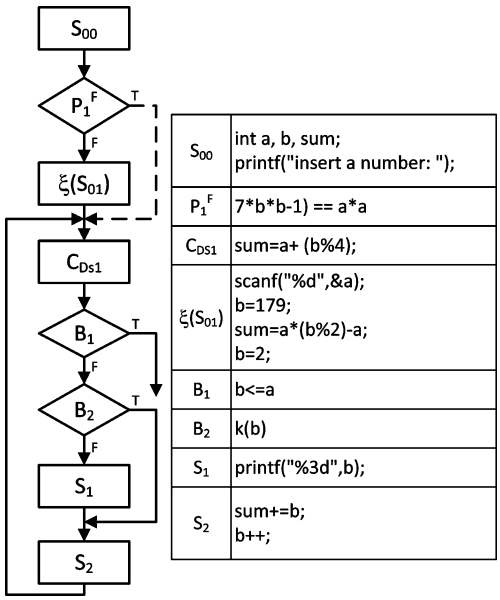

Fig. 11. Program III: the obfuscated result of Program I after applying $\mathcal{T}_2$.

C. Evaluation

Comparing ψ, ψ_1, and ψ_2, the common subgraphs of ψ, ψ_1, and ψ ψ_2 have identical edges $(B_1, B_2)^T$, $(B_1, \phi)^F$, $(B_2, S_1)^T$, $(B_2, S_2)^F$, (S_1, S_2). We have $\sum_i |edge(CS_i(\psi, \psi_1))| = \sum_i |edge(CS_i(\psi, \psi_2))| = 4$. Since the number of edges in ψ, ψ_1, and ψ_2 are 6, 12, and 10,

the distances between these graphs are $d(\psi, \psi_1) = 5/9$ and $d(\psi, \psi_2) = 1/2$.

In ψ, there are two branches B_1 and B_2 with range values 4 and 1. Hence, we obtain $ns(\psi) = 1/2$. In ψ_1, the range values of P_1^T, B_1, B_2, and P_2^F are 4, 5, 1 and 1. So, we obtain $ns(\psi_1) = 11/20$ and $p(\psi, \psi_1) = 1/10$. Similarly, in ψ_2, the range values of P_1^F, B_1, and B_2 are 1, 5, and 1. So we can obtain $ns(\psi_2) = 7/15$ and $p(\psi, \psi_2) = -1/15$.

A positive potency value implies that $\mathcal{T}_1$ achieves obscurity from the perspective of the ratio of NC and R, while a negative value indicates that $\mathcal{T}_2$ contributes nothing to that ratio. With the distance computed by using the proposed MGE, we can obtain two DP vectors $DP(\psi, \mathcal{T}_1) = (5/9, 1/10)$ and $DP(\psi, \mathcal{T}_2) = (1/2, -1/15)$ to show the abilities against reverse engineering provided by $\mathcal{T}_1$ and $\mathcal{T}_2$. Since both distance and potency of ψ_1 are larger than those of ψ_2. We conclude that $\mathcal{T}_1$ provides the better robustness than $\mathcal{T}_2$. The space penalty caused by $\mathcal{T}_1$ can be estimated as $(\bar{S} + 2 \cdot \bar{B})$, where $O_S^{S,2}(\cdot, S_0)$ results in no overheads, $O_D^S(\cdot, B_1)$ leads to $\bar{S}$, and $O_{Op}^T(\cdot, B_1)$ and $O_{Op}^F(\cdot, S_{01})$ lead to two $\bar{B}$s. The space penalty caused by $\mathcal{T}_2$ is $\bar{B} + \bar{S}$, where $O_S^{S,2}(\cdot, S_0)$ and $O_E(\cdot, S_{01})$ do not derives any overheads, but $O_{Op}^F(\cdot, S_{01})$ and $O_D^S(\cdot, B_1)$ lead to $\bar{B}$ and $\bar{S}$, respectively.

IX. Conclusion

We presented a framework for representing, evaluating, and analyzing control-flow obfuscating transformations. We showed that with a graph-based representation, many existing control-flow transformations can be represented as a composition of atomic operators. We have also proposed a new metric for quantifying the effects of these transformations upon the program. Such a metric may help evaluate the robustness and costs of control-flow obfuscating transformations.

Nevertheless, if we consider the side effects of obfuscating a software program—code size will increase and execution performance will slow down—more studies will be needed on how to best compromise between security and performance overheads. We hope that our formal model will provide a helpful framework for examining these tradeoffs in greater depth.

References

[1] S. Subramanya and B. Yi, "Digital rights management," *IEEE Potentials*, vol. 25, no. 2, pp. 31–34, Mar./Apr. 2006.

[2] Y. Nishimoto, A. Baba, T. Kurioka, and S. Namba, "A digital rights management system for digital broadcasting based on home servers," *IEEE Trans. Broadcast.*, vol. 52, no. 2, pp. 167–172, Jun. 2006.

[3] D. Aucsmith, "Tamper-resistant software: An implementation," in *Proc. 1st Int. Workshop Informaiton Hiding*, 1996, vol. 1174, pp. 317–333.

[4] T. Wilkinson, D. Hearn, and S. Wiseman, "Trustworthy access control with untrustworthy web servers," in *Proc. Computer Security Applications Conf.*, Jan. 1999, pp. 12–21.

[5] N. Komninos, B. Honary, and M. Darnell, "Security enhancements for A5/1 without loosing hardware efficiency in future mobile systems," in *Proc. 3rd Int. Conf. 3G Mobile Communication Technologies*, 2002, pp. 324–328.

[6] T. Hou, H. Chen, and M. Tsai, "Three control flow obfuscation methods for Java software," *Proc. Inst. Elect. Eng. Softw.*, vol. 153, no. 2, pp. 80–80, Jan. 2006.

[7] M. D. Preda and R. Giacobazzi, "Control code obfuscation by abstract interpretation," in *Proc. 3rd IEEE Int. Conf. Software Engineering and Formal Methods*, 2005, pp. 301–310.

[8] D. Low, "Java control flow obfuscation," M.Sc. dissertation, Univ. Auckland, Auckland, New Zealand, 1998.

[9] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Rep. no. 148, 1997, Univ. Auckland Tech. Rep..

[10] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, Aug. 2002.

[11] J. Memon, S. u. Arfeen, A. Mughal, and F. Memon, "Preventing reverse engineering threat in Java using byte code obfuscation techniques," in *Proc. Int. Conf. Emerging Technologies*, 2006, pp. 689–694.

[12] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," *Foundations Intrus. Tolerant Syst.*, pp. 273–282, 2003.

[13] T. Toyofuku, T. Tabata, and K. Sakurai, "Program obfuscation scheme using random numbers to complicate control flow," Jan. 2005, IEIC Tech. Rep.

[14] T. László and A. Kiss, "Obfuscating C++ programs via control flow flattening," *Annales Universitatis Scientiarum de Rolando Eötvös Nominatae—Sectio Computatorica*, pp. 15–15, May 2008.

[15] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation," in *Proc. 5th ACM Workshop on Digital Rights Management*, Nov. 2005, pp. 83–92.

[16] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proc. 16th USENIX Security Symp.*, 2007, pp. 275–290.

[17] B. A. Cota and R. G. Sargent, "Automatic Lookahead computation for conservative distributed simulation," CASE Center Tech. Rep. 8916, 1989.

[18] B. Cota, D. Fritz, and R. Sargent, "Control flow graphs as a representation language," in *Proc. Simulation Conf.*, 1994, pp. 555–559.

[19] "Chilling effects clearinghourse." Dec. 12, 2008. [Online]. Available: http://www.chillingeffects.org.

[20] H. Bunke and K. Shearer, *A Graph Distance Metric Based on the Maximal Common Subgraph*, vol. 19, no. 3-4, pp. 255–259, Mar. 1998.

[21] W. Wallis, P. Shoubridge, M. Kraetz, and D. Ray, "Graph distances using graph union," *Pattern Recogn. Lett.*, Jan. 2001.

[22] H. Zuse, *Software Complexity: Measures and Methods*. Hawthorne, NJ: Walter de Gruyter Co., 1991.

Hsin-Yi Tsai received the B.S. and M.S. degrees in electrical and control engineering from the National Chiao-Tung University, Taiwan, in 2007, where she is currently pursuing the Ph.D. degree in electrical and control engineering.

Her research interests include wireless security, DRM protection, and security analysis.

Ms. Tsai is a member of the Phi Tau Phi Society since 2007.

Yu-Lun Huang received the B.S. and Ph.D. degrees in computer science and information engineering from the National Chiao-Tung University, Taiwan, in 1995 and 2001, respectively.

Currently, she is an Assistant Professor in Department of Electrical and Control Engineering of National Chiao-Tung University. Her research interests include wireless security, secure testbed design, embedded software, embedded operating systems, network security, secure payment systems, VoIP, and QoS.

Dr. Huang has been a member of the Phi Tau Phi Society since 1995.

David Wagner is an Associate Professor in the Computer Science Division at the University of California at Berkeley, working in the areas of computer security and electronic voting. He has studied the security of cell-phone standards, 802.11 wireless networks, electronic voting systems, and other widely deployed systems, and is active in the areas of software security and systems security.

Prof. Wagner is a past CRA Digital Government Fellow and Alfred P. Sloan Research Fellow. He received an Honorable Mention in the ACM Doctoral Dissertation Award competition for his Ph.D. work.