Available online at www.sciencedirect.com
SciVerse ScienceDirect
journal homepage: www.elsevier.com/locate/cose
Computers & Security

A framework for quantitative evaluation of parallel control-flow obfuscation

Yu-Lun Huang*, Hsin-Yi Tsai

Institute of Electrical Control Engineering, National Chiao Tung University, Taiwan

ARTICLE INFO

Article history:
Received 29 August 2011
Received in revised form
8 May 2012
Accepted 8 August 2012

Keywords:
Software obfuscation
Security evaluation
Reverse engineering
Software metrics
Control flow graph

ABSTRACT

Software obfuscation is intended to protect a program by thwarting reverse engineering. Several types of software obfuscation have been proposed, and control-flow obfuscation is a commonly adopted one. In this paper, we present a framework to evaluate parallel control-flow obfuscation, which raises difficulty of reverse engineering by increasing parallelism of a program. We also define a control flow graph of a program and some atomic operators for obfuscating transformations. The proposed framework comprises three phases: parsing, formalization and evaluation. A program is first parsed to a control flow graph. Then, we formalize a parallel control-flow obfuscating transformation based on our atomic operators. By selecting target code blocks in the control flow graph and applying obfuscating transformations to the target code blocks, the original program is then obfuscated. In the third phase, we define a measure to calculate the program complexity. The measure can be considered as a degree to which an obfuscating transformation can confuse a human trying to understand the obfuscated program. Such a measure can also be used as the base of the potency metric to estimate the capability of the obfuscated program against reverse engineering. Our novel framework helps efficiently examine a control-flow obfuscating transformation in a systematic manner and helps select an appropriate obfuscating transformation among a number of candidates to better protect a program.

1. Introduction

Software obfuscation is a technique used to increase potency of a program and protect the software piracy. Namely, software obfuscation transforms a program into an obfuscated one which thwarts reverse engineering, but still preserves the original functionality (Collberg et al., 1997). Despite a theoretic proof of impossibility of omnipotent software obfuscation (Barak et al., 2001), positive results can still be obtained under specific conditions (Lynn et al., 2004).

According to Collberg's study (Collberg et al., 1997), software obfuscation can be classified into three types: control-flow obfuscation, data obfuscation and layout obfuscation. Control-flow obfuscation disguises the real execution of a program under the scrambled control flow of that program to make reverse engineering difficult. Data obfuscation transforms data and data structures in a program. Layout obfuscation removes information that an attacker can seize from a program. Various obfuscating transformations of each type have been proposed (Collberg et al., 1997; Wang et al., 2003; Popov et al., 2007; Majumdar et al., 2007b; Kazuhide et al., 2006). Each obfuscating transformation has its individual restrictions and provides different levels of resistance against reverse engineering. Therefore, some methods were

* *Corresponding author*. Tel.: +886 988 227 966; fax: +886 3 5715998.
E-mail addresses: ylhuang@cn.nctu.edu.tw, huang.yulun@gmail.com (Y.-L. Huang), hytsai.ece96g@nctu.edu.tw (H.-Y. Tsai).
0167-4048/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cose.2012.08.003

conceived to measure the capability of an obfuscating transformation (Udupa et al., 2005; Madou et al., 2006; Naeem et al., 2007; Anckaert et al., 2007; Majumdar et al., 2007a; Ceccato et al., 2008, 2009; Tsai et al., 2009). The evaluation methods can further be classified into two types: empirical analysis and formal analysis.

The empirical analysis evaluates an obfuscating transformation by running practical experiments to obtain evaluation results (Udupa et al., 2005; Madou et al., 2006; Naeem et al., 2007; Anckaert et al., 2007; Majumdar et al., 2007a), or to determine the time and effort a human takes to interpret an obfuscated program (Ceccato et al., 2008, 2009). The above work (Udupa et al., 2005; Madou et al., 2006; Naeem et al., 2007; Anckaert et al., 2007; Majumdar et al., 2007a; Ceccato et al., 2008, 2009) executed various experiments to manifest the effects and capabilities of different obfuscating transformations.

In the empirical studies, implementation of new transformations is required to repeat the experiments on applying different transformations to the same program. The experiment results show that effects stemming from an obfuscating transformation may change with the structures of a target program where the transformation is applied to. Moreover, different combinations of obfuscating transformations may produce different results. In short, an experiment result is tightly related to a specific program with a specific obfuscating transformation. It is less efficient and inflexible to estimate the capability of an obfuscating transformation by the empirical studies, especially when we want to compare multiple transformations and select a proper one to protect a designate program at the design stage.

In our previous work (Tsai et al., 2009), we designed a framework to quantify the effects of control-flow obfuscating transformations. We proposed a model to formalize the obfuscating transformations and target programs. Then, we devised measures to assess the capability of the transformations. Based on our model and measures, we can compare different control-flow obfuscating transformations easily. The capability of a control-flow obfuscating transformation can be estimated at the design stage on the basis of the formal representation of the transformation. Nevertheless, our previous work can only measure the capability of a control-flow obfuscating transformation when applied on a sequential program. Since parallel programs are popular today, we extend our previous framework to evaluate the effect of an obfuscating transformation.

This paper presents a framework for evaluating parallel control-flow obfuscating transformations, which aim at scrambling the control-flow graph (CFG) of a program by increasing its parallelism. The framework consists of three components: parser, formalizer and evaluator. The **parser** is in charge of interpreting and converting a program, either parallel or sequential, to its corresponding CFG. The **formalizer** can be used to describe control-flow obfuscating transformations, which can be decomposed into the defined atomic operators. We devise a new atomic operator to complement the insufficiency of our previous work (Tsai et al., 2009), such that parallel control-flow obfuscating transformations can be modeled by our framework. After applying a transformation to a specified CFG, either parallel or sequential, we can obtain an obfuscated CFG. Then, the **evaluator** analyzes the capability of an obfuscating transformation by comparing the original and obfuscated CFGs. We present a new measure to determine complexity of a parallel program, which is a base for deriving the capability of a parallel control-flow obfuscating transformation. We conjecture that the measure may be related to the capability of an obfuscated program against reverse engineering.

We hope that based on the framework we can not only efficiently examine a control-flow obfuscating transformation but easily understand how to assemble and create a more effective transformation, especially at the design stage. We also expect that our framework, concerning the impacts resulting from choosing different target code blocks, helps select targets and proper obfuscating transformations to reach better potency.

This paper is organized as follows. In Section 2, we review the related work. Section 3 introduces the definition of a parallel CFG and Section 4 describes formalization of parallel control-flow obfuscating transformations. Section 5 presents a measure for evaluating parallel control-flow obfuscating transformations. We demonstrate our framework by examples in Section 6 and conclude the paper in the last section.

2. Related work

In recent years, many researchers have proposed various evaluation methods to assess effectiveness of an obfuscating transformation. The methods coarsely fall into two types: empirical analysis (Udupa et al., 2005; Anckaert et al., 2007; Majumdar et al., 2007a; Ceccato et al., 2008, 2009) and formal analysis (Tsai et al., 2009). The empirical analysis evaluates an obfuscating transformation by running practical experiments to observe how much an obfuscated program resists against deobfuscators or how much time a human takes to interpret the program. The formal analysis mainly focuses on modeling an obfuscating transformation into a formal representation and measures the transformation on the basis of the formalization.

2.1. *Empirical analysis*

Udupa et al. (2005) examined control-flow flattening, a control-flow obfuscating transformation, by measuring the time required for automatic deobfuscation. Anckaert et al. (2007) introduced a framework to evaluate an obfuscating transformation based on program complexity metrics, which measure the complexity with respect to instructions, control flow, data flow and data. The authors implemented three obfuscating transformations (control flow flattening, static disassembly thwarting and binary opaque predicates), applied the transformations to eleven C programs of the SPECint 2000 benchmark suite, and produced obfuscated programs. The complexity analyses show that the three transformations can provide positive effects, but the ‘binary opaque predicates’ transformation is less potent than the other two transformations. Majumdar et al. (2007a) introduced several metrics to evaluate the effect of another reverse-engineering technique, backward slicing. The authors also implemented three obfuscating transformations (bogus predicate, adding to a while loop and variable encoding), applied them to five programs and obtained the differences of metric values before and after obfuscation. The differences imply that these obfuscating transformations can significantly make reverse engineering difficult.

Taking questionnaires as a basis, Ceccato et al. (2008, 2009) assessed the difficulty an attacker would encounter in examining the ‘identifier renaming’ obfuscation technique. The authors asked human users to interpret the original and obfuscated programs and fill out a questionnaire. After analyzing the questionnaires with famous statistical tests, such as the Mann–Whitney test and the Wilcoxon test, the authors pointed out that the ‘identifier renaming’ technique can effectively reduce the capability of human users in understanding the source codes of an obfuscated program.

The above empirical studies (Udupa et al., 2005; Anckaert et al., 2007; Majumdar et al., 2007a; Ceccato et al., 2008, 2009) intended to understand the effects of obfuscation. They launched practical experiments to measure individual obfuscating transformations with the defined metrics or with the perception of human users. These experiment results reflected the relation between a specific program and an obfuscating transformation. Since a transformation may cause different effects upon different programs, it is necessary to assess obfuscation in finer grains, such that the differences can be distinguished. With a fine grained assessment, we can also evaluate effect of a compound obfuscating transformation which is constituted by one or more individual transformations. However, the empirical studies provide little knowledge to address the issues, like the order of transformations or the possibility of obtaining different results even if we apply the same transformation to the same program.

2.2. *Formal analysis*

In our previous work (Tsai et al., 2009), we presented a graph approach to quantifying the effect of control-flow obfuscating transformations. We proposed a formal method based on atomic operators which control-flow obfuscating transformations can be decomposed into. Each atomic operator is assigned a code block as its target such that an obfuscating transformation can be formally modeled in finer grains. Table 1 lists the nine atomic operators defined in our previous work.

Since different sequences of atomic operators may lead to different obfuscating transformations, our previous work is helpful in understanding the effect of a compound transformation and the effect caused by different order of transformations. The work also considers that the effect of an obfuscating transformation may change with target code blocks even if the atomic operators are applied in the same sequence. However, the method was designed simply for sequential control-flow obfuscating transformations. As an improvement of the programming skills, it is essential to evaluate parallel control-flow obfuscating transformations, in addition to the sequential transformations. To provide a more comprehensive evaluation, this paper improves our previous work to support the capability of assessing parallel control-flow obfuscating transformations.

Table 1 – Atomic operators of control-flow obfuscating transformations (Tsai et al., 2009).

Atomic operators	Notations
Insert type I/II/III opaque predicates	$O^F_{Op}/O^T_{Op}/O^?_{Op}$
Split simple code blocks	$O^{S,n}_S$
Split branches	$O^{B,n}_S$
Reorder code blocks	O_R
Replace with equivalent codes	O_E
Insert dummy simple blocks	O^S_D
Insert dummy loops	O^L_D

3. Parsing phase

Control flow graphs (CFGs) can be used to represent the control flows of a program and to help an analyzer understand the program easily (Cota et al., 1994; Stotts and Cai, 1988; Cheng, 1993). In this paper, we facilitate the formalization of parallel control-flow obfuscating transformations by leveraging the definitions of a CFG. As a high-level abstraction, a program can be parsed into a directed graph whose vertices are code blocks of the program and edges represent the execution sequence of two adjacency nodes. An edge between two code blocks, ‘X’ and ‘Y’, implies that the code block ‘Y’ should be executed immediately after ‘X’.

In this paper, the vertices (code blocks) in a parallel CFG can be classified as:

- Branch (B): a code block that causes execution to transfer, either conditionally or unconditionally, to some statement other than the succeeding statement. In high-level programming languages, branch instructions may be found in for, while, do-while, if-else, and goto statements.
- Fork (F): a code block that creates parallel execution. The succeeding code blocks of a fork can run concurrently until the paths converge.
- Join (J): a code block at which parallel execution paths converge.
- Simple block (S): a code block with an ordered sequence of statements without any outgoing or incoming **branch**, **fork** or **join** statements inside the code block.

In addition, we also adopt the notations defined in our previous work to mention some special code blocks.

- Entry block (C^E) : the entry code block of a source program.
- Any block (C^A) : any existing code block.
- Dummy code block (C^D) : a code block running dummy instructions that do not affect the final execution result of a program.
- Termination block (ϕ): the exit code block of a source program.

As mentioned, edges in a parallel CFG represent possible execution paths the program may take. Our parser specifies four types of edges for illustrating a parallel CFG.

- Sequential edge (s): A sequential edge $s = (C_i, C_j)$ exists between two code blocks C_i and C_j, where $C_i \in \{S, J\}$ and $i \neq j$.
- Branch edge (b): Since a branch B may jump to either its true or false target, there are two code blocks that could be executed immediately after B. Hence, the two branch edges leaving B are denoted by $b^T = (B, C^{True})^T$ and $b^F = (B, C^{False})^F$, where C^{True} and C^{False} represent the true and false targets of B, respectively.
- Fork edge (f): Since several code blocks can be executed concurrently right after a fork F, a fork block F may have one or more succeeding code blocks. A fork edge is therefore represented as $f = (F, C)$, where $C \neq J$.
- Join edge (j): A join edge is denoted by $j = (C, J)$, where $C \neq F$. Such an edge means that the execution of C is always followed by immediately executing J.

With the above definitions, a directed graph G is defined as $G = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the vertex set and $\mathbb{E}$ represents the edge set. $\mathbb{V}$ contains all the code blocks of the parsed program, including simple blocks, branches, forks, joins and a termination. $\mathbb{E}$ is composed of sequential edges, branch edges, fork edges and join edges. Then, a parsed program ψ is represented as (C^E, G), where C^E points to the entry block and G is the CFG of the program. In Fig. 1, the CFG of ψ contains four simple blocks (rectangular), one branch (diamond), one fork (base-down triangular) and one join (base-up triangular). Hence, the formal form of ψ can be represented as $\psi = (S_0, G)$, where $G = (\mathbb{V}, \mathbb{E})$, $\mathbb{V} = \{S_0, S_1, S_2, S_3, S_4, B_0, F_0, J_0, \phi\}$, and $\mathbb{E} = \{(S_0, F_0), (F_0, B_0), (F_0, S_3), (B_0, S_1)^T, (B_0, S_2)^F, (S_1, S_2), (S_2, J_0), (S_3, J_0), (J_0, \phi)\}$. ϕ is an indication of the end of execution path and is not counted into the number of vertices in $\mathbb{V}$.

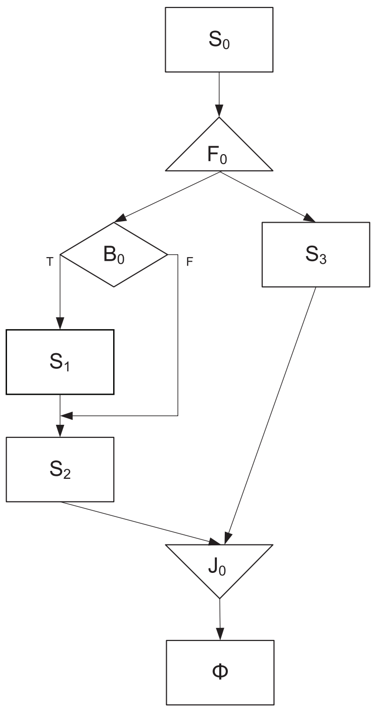


Fig. 1 – Example of a parallel control flow graph.

4. Formalization phase

A parallel control-flow obfuscating transformation converts a program into a parallel program with higher level of parallelism. Though the nine atomic operators defined in our previous work can be applied to a parallel program for higher parallelism, these operators cannot make a sequential program become a parallel program. Hence, the scalability of the framework is restricted to model sequential obfuscating transformations only. Hence, we design an additional atomic operator "inserting a fork" to complement the insufficiency. The new atomic operator can be used to create forks and fork edges for complicated control flows. Together with the newly designed atomic operator, the new set of atomic operators can provide the basic components to formalize a parallel control-flow obfuscating transformation.

Fig. 2 shows the atomic operator of inserting a fork. The operator, denoted by $O_F^n(\psi, C^T)$, indicates that a fork is inserted as the predecessor of n consequent code blocks, $C_k, C_{k+1}, \cdots, C_{k+n-1}$. C^T is used to index the above target code block C_{k+i} $(0 \leq i \leq n-1)$.

After executing the operator, $C_k, C_{k+1}, \cdots, C_{k+n-1}$ are executed in parallel immediately after C_x, the successor of F_1. In addition to inserting a fork, we insert a join (J_1) to guarantee that the execution of these parallel code blocks is completed before executing C_y (the successor of J_1), such that the original functionality of the program can be preserved. Algo. 4 clarifies the steps O_F takes.

Before applying O_F, the execution dependency among C_k and its successors $(C_{k+1}, C_{k+2}, \cdots, C_{k+n-1})$ should be checked. If dependency exists, O_F may result in an incorrect execution. Another restriction on applying O_F is that the target code blocks of such an operator cannot be a join, a branch or any other fork.

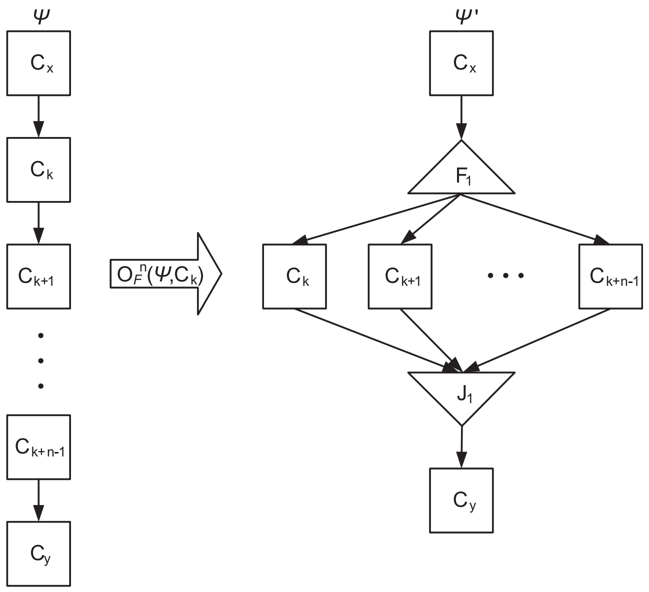


Fig. 2 – The operation of inserting a fork code block. After insertion, $\mathbb{E}$ becomes $\{(\mathbf{C_x}, \mathbf{F_1}), (\mathbf{F_1}, \mathbf{C_k}), (\mathbf{F_1}, \mathbf{C_{k+1}}), \ldots, (\mathbf{F_1}, \mathbf{C_{k+n-1}}), (\mathbf{C_k}, \mathbf{J_1}), \ldots, (\mathbf{C_{k+n-1}}, \mathbf{J_1}), (\mathbf{J_1}, \mathbf{C_y})\}$.

Algo. 4. Insert a fork on n consequent code blocks, $O_F^n(\psi, C^T),\ C^T = C_K$.

```
$\mathbb{V} \leftarrow \mathbb{V} \cup \{F, J\}$;
$\mathbb{E} \leftarrow \mathbb{E} \cup \{(F, C_k), (F, C_{k+1}), \cdots, (F, C_{k+n-1})\}$;
Find $(C_{k+n-1}, C_i)$ and insert $(J, C_i)$ to $\mathbb{E}\ \forall i$;
IF $C_k = C^E$ THEN
    Replace the entry point with $F$;
END IF;
Replace $e_i = (C_i, C_k)$ with $(C_i, F)\, \forall e_i \in \mathbb{E}$;
Replace $e_i = (C_i, C_j)$ with $(C_i, J)\, \forall C_i \in \{C_k, C_{k+1}, \cdots, C_{k+n-1}\}$;
```

The new atomic operator O_F and the existing operators listed in Table 1 can form a set of the building components to formalize parallel control-flow obfuscating transformations. With such a set of atomic operators, each transformation can be represented as $\mathcal{T} = \langle f_1, f_2 \cdots f_m \rangle$, a composition of m atomic operators $f_1, \ldots, f_m$, where $f_x \in \{O_{Op}^F(\cdot, C_a), O_{Op}^T(\cdot, C_b), O_{Op}^?(\cdot, C_c), O_E(\cdot, C_d), O_S^{S,n}(\cdot, C_e), O_S^{B,n}(\cdot, C_f), O_D^L(\cdot, C_g), O_D^S(\cdot, C_h), O_R(\cdot, C_i), O_F^n(\cdot, C_j)\}$, for $x = 1, \ldots, m$. $\langle\rangle$ denotes an ordered set of functional composition, where $\langle f_1, f_2 \cdots f_m \rangle$ represents a function applying multiple atomic operators to a target program, i.e., $f_m(\cdots f_2(f_1(target)))$. C_a, $C_b, \cdots$, C_j are arbitrary code blocks of a source program.

With the above definition, we can formally describe a parallel control-flow obfuscating transformation, which is helpful for further evaluation. In our definition, since we not only take a program ψ as an input of a transformation, but also consider distinct target code blocks for each atomic operator, a finer grained evaluation result can be obtained.

5. Evaluation phase

To measure the complexity and overhead of an obfuscated program, Collberg et al. (1997) proposed three metrics to evaluate the performance of an obfuscating transformation. They proposed a cost metric to measure the additional runtime resources required to execute an obfuscated program. The resilience metric has been introduced to evaluate how well an obfuscating transformation holds up against attacks from an automatic deobfuscator. The potency metric is supposed to estimate 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 exact effort of reverse engineering is difficult to quantify and estimate, due to the variance of individual experience and skill. It may take some hackers significantly longer than others to reverse engineer the same program. In this paper, our evaluation approach tries to eliminate the individual variances. Assuming that the effort of reverse engineering a program is the same to all hackers, the proposed measure can then be defined based on the potency metric given by Collberg et al. (1997):

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

where $comp(PG)$ and $comp(PG')$ stand for the complexity of the original program PG and the obfuscated program PG', respectively.

5.1. *Complexity measure*

To calculate the potency that a parallel control-flow obfuscating transformation contributes, we first need an appropriate complexity measure that can reflect the effects resulting from the transformation. Since our framework considers both sequential and parallel programs, the complexity measure needs to take both sequential and parallel programs into account.

Shatz (1988) suggested a framework that measures the complexity of a distributed program. Such a distributed program consists of several processes executing asynchronously in parallel but communicating synchronously by passing messages. In Shatz's opinion, a distributed program's complexity can be obtained based on the complexity of each process and complexity stemming from interaction between the processes. A parallel program, similar to a distributed program, is composed of a number of sequential programs which are executed in parallel. Hence, the total complexity of a parallel program plausibly compromises the 'sequential complexity' that its sequential programs contribute and the complexity resulting from 'level of parallelism'. Therefore, we define the total complexity of a parallel program (ψ) as shown in Eq. (2):

$$comp(\psi) = w_s \times scomp(\psi) + w_p \times pcomp(\psi), \tag{2}$$

where $comp(\psi)$, the complexity of ψ, is comprised of two parts: sequential complexity $scomp(\psi)$ and complexity resulting from parallelism $pcomp(\psi)$. In this equation, w_s and w_p are defined as adjustable weights for setting the ratio between the two types of complexities.

5.2. *Sequential complexity*

To prevent against reverse engineering, a control-flow obfuscating transformation makes a program's control flow unintelligible by increasing depth of nests in a program or by enlarging the size of a program. Therefore, a proper complexity measure should calculate the complexity of an obfuscated program in consideration of the effects a control-flow obfuscating transformation poses.

In 1981 Harrison and Megal presented the measure SCOPE (Zuse, 1991) to calculate complexity of a control flow graph of a sequential program. SCOPE determines the complexity in terms of the size and the depth of nests involved in a control flow graph. Since transforming a CFG into another by control-flow obfuscation usually leads to changes of the graph size or the nesting level, SCOPE can be an appropriate indicator for estimating the potency. Therefore, we adopt the definition of SCOPE for evaluating the capability of control-flow obfuscation, instead of taking other measures, such as Relative Logical Complexity (RLC) and Absolute Logical Complexity (ALC) (Zuse, 1991).

As mentioned in Section 3, we can obtain a directed CFG (ψ) after parsing a source program. Then, we can derive the scope value of ψ by Eq. (3).

$$scope(\psi) = \sum_{B_i \in \mathbb{B}} |range(\psi, B_i)|, \tag{3}$$

where $\mathbb{B}$ is the set of branches in ψ and $|range(\psi, B_i)|$ stands for the nesting levels that B_i contributes. In this equation, $|range(\psi, B_i)|$ represents the number of code blocks either in a loop led by B_i or on the paths branching out from B_i to the convergence of the paths. The value of *scope* increases as the number of nodes in the nests of a program increases. Moreover, if SCOPE is adopted in Eq. (2), the SCOPE value can be derived by summing up the complexities of sub graphs of the CFG (ψ). This simplifies the analyzing process of a complicated CFG.

In addition to the nesting level, the number of condition expressions within a branch also contributes complexity of a program. The more the condition expressions are, the more efforts should be taken to analyze the control flow. Hence, we extend the definition of SCOPE to contemplate the impacts brought by the number of condition expressions in a branch.

From a high-level programming perspective, a branch can be treated as a building block for two types of control flow structures, conditional jump and loop, which pose individual effects upon complexity.

5.2.1. *Conditional jump*

The fundamental control flow graph of a conditional jump is shown in Fig. 3. In this figure, the program contains a branch block with two target code blocks converging at C^A. Assuming that the branch B_i contains only one condition expression, we can obtain $|range(\psi, B_i)| = 2$. In reality, a branch normally can contain more than one condition expressions. So, we need to consider such a nesting situation as well. If B_i contains n_c condition expressions with one or more "AND" or "OR" operators, then B_i can be split into n_c branches (says $B_{i1}, B_{i2} \cdots Bi_{(n_c)}$), each of which contains only one condition expression. The new flow graph deduced from Fig. 3 is now extended to n_c branches and 3 simple code blocks. The nesting level of the branch B_{i1}, in the deepest nest, is equal to 2 since two code blocks C^{True} and C^{False} are located on the divergent paths branching out at B_{i1}. For branch B_{i2}, 3 code blocks (B_{i1}, C^{True} and C^{False}) are likely executed, such that we can derive $|range(\psi, B_{i2})| = 3$. By induction, the range value of branch B_{ij} can be deduced as $|range(\psi, B_{ij})| = |range(\psi, B_{i1})| + j - 1$.

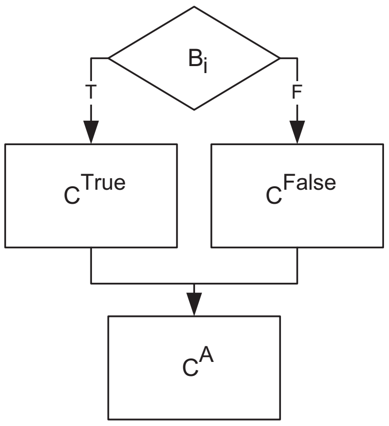


Fig. 3 – The control flow graph of a conditional jump.

Considering the above case that a branch (says B_i) contains n_c condition expressions, we define the compound range of the branch in Eq. (4).

$$\begin{aligned}|crange(\psi, B_i)| &= \sum_{j=1}^{n_c} |range(\psi, B_{ij})| = \sum_{j=1}^{n_c} (|range(\psi, B_{i1})| + j - 1) \\ &= \sum_{j=1}^{n_c} (|range(\psi, B_i)| + j - 1) \\ &= n_c \times |range(\psi, B_i)| + \sum_{j=2}^{n_c} (j - 1).\end{aligned} \tag{4}$$

5.2.2. *Loop*

In a high-level program, a loop may be created by for, while and do-while statements. Fig. 4 shows the control flow graph of a loop, which produces one branch and two simple code blocks. If B_i in the loop contains only one condition expression, the nesting level, denoted by $|range(\psi, B_i)|$, equals to 2 since two code blocks (C^{True} and B_i) are located on the paths branching out at B_i. Similarly, if B_i contains n_c condition expression, we said that B_i can be further split into n_c pieces ($B_{i1}, B_{i2}, \cdots, B_{i(n_c)}$), as illustrated in Fig. 5. Then, we can come out a new CFG containing two simple code blocks and n_c branch blocks. In such a loop, since all branches and C^{True} are executed before the divergent paths meet at C^A, the number of the code blocks on the divergent paths becomes n_c+1. Hence, we obtain $|range(\psi, B_{ij})| = n_c + 1 = n_c + |range(\psi, B_i)| - 1,\ \forall 1 \le j \le n_c$.

Again, by induction, the compound range of B_i, containing n_c condition expressions, can be derived as

$$|crange(\psi, B_i)| = n_c \times (|range(\psi, B_i)| + n_c - 1) \tag{5}$$

5.3. *Level of parallelism*

Recall that the total complexity of a program is contributed by the sequential complexity and the level of parallelism of a program. In the equation (Eq. (2)) calculating the total

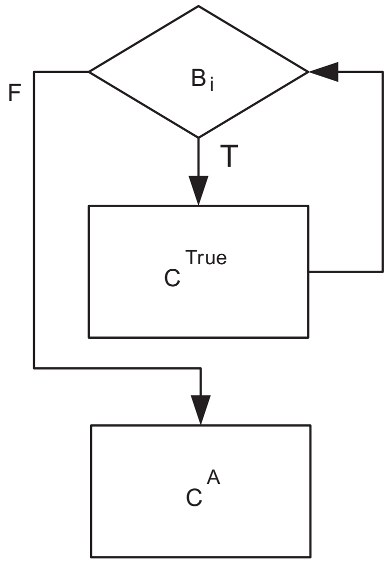


Fig. 4 – The control flow graph of a loop.

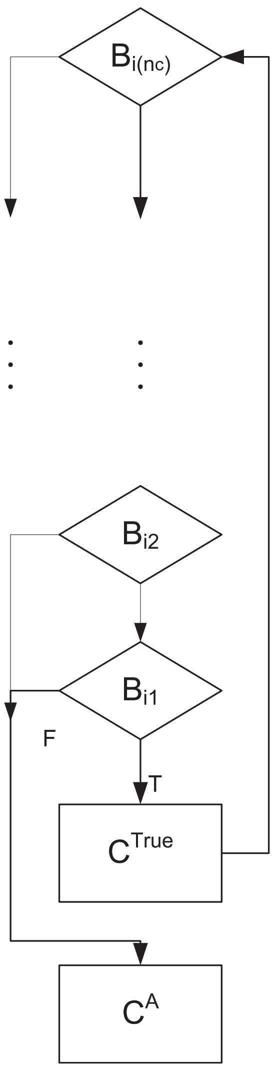


Fig. 5 – The extended control flow graph that *B* contains i condition expressions.

complexity of a program, we estimate the sequential complexity (*scomp*) by adopting SCOPE. For the complexity resulting from the level of parallelism (*pcomp*), we assume that the number of code blocks executed in parallel contributes to the level of parallelism of a program (ψ). We then extend the Shatz's concept to define $pcomp(\psi)$, as expressed in Eq. (6):

$$pcomp(\psi) = \sum_{F_i \in \mathbb{F}} |range(\psi, F_i)|, \tag{6}$$

where $\mathbb{F}$ is the set of forks in ψ and $|range(\psi, F_i)|$ represents the parallelism that the fork F_i is conducive to. $|range(\psi, F_i)|$ indicates the number of code blocks on the parallel execution paths, which start from F_i and terminate at the join operation. If no forks exist in ψ, i.e. ψ is a sequential program, then $pcomp(\psi) = 0$.

In this paper, we propose a measure to estimate the complexity of a program and the capability of a control-flow obfuscating transformation. We recognize the measure merely serves as a heuristic indicator of security. However, we show our first attempt at evaluating a parallel control-flow obfuscating transformation in a methodical manner, and we believe the measure can still be the first step toward evaluation of software potency raised by a control-flow obfuscating transformation. We do not claim that a large value of our measure implies that an obfuscated program will necessarily be secure against reverse-engineering. We expect that large values of this measure are necessary but not sufficient for security. Our measure 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, this measure may still be helpful in evaluating a control-flow obfuscating transformation, especially in comparing different obfuscating transformations when applied to a single program.

6. Example: prime number generator

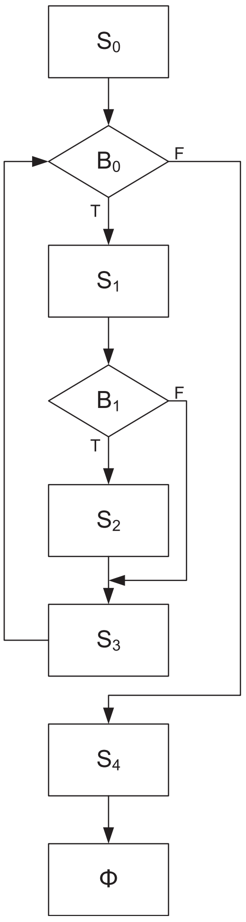
In this section, we show how the proposed framework can be used to evaluate the capability of obfuscating transformations on a given software program (a prime number generator in this example).

6.1. *CFG conversion*

Program I, generating prime numbers smaller than an input *num*, is used as an example for demonstration.

```
/* Program I. Prime number generator */
int _PrimeGen (int tmp) {
  int i;
  for (i=2; i<=tmp/2; i++)
   if (tmp %i = = 0)
   return 0;
    return 1;
}
int main () {
  int num, tmp, sum;
  int PrimeOrNot=0;
  printf("insert a number \n");
  scanf("%d", &num);
  for(sum=0, tmp=2; tmp<=num; tmp++) {
   PrimeOrNot = _PrimeGen(tmp);
   if( PrimeOrNot = = 1)
   printf("%6d", tmp);
   sum += tmp;
     }
     printf("\n");
     return 0;
}
```

We derive the CFG ψ of Program I, as shown in Fig. 6. ψ contains 2 branches and 5 simple code blocks. According to the equations defined in Section 5, we get $scomp(\psi) = |range(\psi, B_0)| + |range(\psi, B_1)| = 5 + 1 = 6$, $pcomp(\psi) = 0$, and $comp(\psi) = 6 + 0 = 6$.



S_0	int num, tmp, sum;
	int PrimeOrNot=0;
	printf("Insert a number: \n");
	scanf("%d",&Num);
	sum=0;
	tmp=2;

B_0	tmp <= num
S_1	PrimeOrNot = _PrimeGen(tmp);
B_1	PrimeOrNot == 1
S_2	printf("%6d",tmp);

S_3	sum += tmp;
	tmp ++;

S_4	printf("\n");
	return 0;

Fig. 6 – CFG of Program I: $\psi = (\mathbf{C^E}, (\mathbb{V}, \mathbb{E}))$, where $\mathbf{C^E} = \mathbf{S_0}$, $\mathbb{V} = \{\mathbf{S_0}, \mathbf{S_1}, \mathbf{S_2}, \mathbf{S_4}, \mathbf{B_0}, \mathbf{B_1}, \phi\}$, and $\mathbb{E} = \{(\mathbf{S_0}, \mathbf{B_0}), (\mathbf{B_0}, \mathbf{S_1})^{\mathbf{T}}, (\mathbf{B_0}, \mathbf{S_4})^{\mathbf{F}}, (\mathbf{S_1}, \mathbf{B_1}), (\mathbf{B_1}, \mathbf{S_2})^{\mathbf{T}}, (\mathbf{B_1}, \mathbf{S_3})^{\mathbf{F}}, (\mathbf{S_2}, \mathbf{S_3}), (\mathbf{S_3}, \mathbf{B_0}), (\mathbf{S_4}, \phi)\}$.

6.2. *Obfuscation and estimation*

In this example, we design two cases to show that different compositions of the atomic operators with different parameters can lead to discriminating effects. We apply a parallel control-flow obfuscating transformation introduced in Collberg's study (Collberg et al., 1997) to the two cases with different target code blocks. In the two cases, we create and insert a dummy code block D_0 into ψ, and then we add a fork block F_0 and a join block J_0 to generate the parallel execution. The first case takes a branch block (B_1) and a simple block (S_1) as the target code blocks; while the second takes the code blocks S_2 and D_0 as targets.

In the first transformation $\mathcal{T}_1$, D_0 is redundant to a branch block B_1 and the fork block F_0 is inserted before S_1, as illustrated in Fig. 7. According to the definitions given in Section 3, $\mathcal{T}_1$ can be formalized as follows:

$$\mathcal{T}_1 = \langle O_D^S(\cdot, B_1), O_F^2(\cdot, S_1)\rangle. \tag{7}$$

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

$$\mathbb{V} \leftarrow \mathbb{V} \cup \{D_0\},$$
$$\mathbb{E} \leftarrow (\mathbb{E} - \{(S_1, B_1)\}) \cup \{(S_1, D_0), (D_0, B_1)\}.$$

We create a dummy block D_0 containing a dummy function *kidfunc()*:

```
void kidfunc (int* t)
{
  int t1=*t;
  while(t1- & t1%10!=0);
}
```

Since D_0 is placed within the loop led by B_0, $|range(\psi, B_0)|$ is increased by 1. Hence, we can derive $scomp(\psi) = |.range(\psi, B_0)|. + |.range(\psi, B_1)|. = 8 + 1 = 9$.

- Running $O_F^2(\cdot, S_1)$:

$$\mathbb{V} \leftarrow \mathbb{V} \cup \{F_0, J_0\},$$
$$\mathbb{E} \leftarrow \left(\mathbb{E} - \left\{(D_0, B_1), (S_1, D_0), (B_0, S_1)^T\right\}\right)$$
$$\cup \left\{(F_0, S_1), (F_0, D_0), (J_0, B_1), (B_0, F)^T, (S_1, J_0), (D_0, J_0)\right\}.$$

After inserting $O_F^2(\cdot, S_1)$, S_1 and D_0 become the immediate successors of F_0 such that S_1 and D_0 can be executed in parallel. By Eq. (6), we obtain $pcomp(\psi) = |range(\psi, F_0)| = 2$, while the sequential complexity remains the same ($scomp = 6$). By assigning the same weights to *scomp* and *pcomp*, we can obtain the total complexity $comp(\psi) = 9 + 2 = 11$.

By Eq. (1), the potency resulted from $\mathcal{T}_1$ is computed as 11/6−1 = 5/6. The positive potency value implies that $\mathcal{T}_1$ obscures the program from the perspective of code complexity.

The second transformation $\mathcal{T}_2$ consists of the same atomic operators as $\mathcal{T}_1$, but applies to different targets. In the second transformation, we first create a dummy block D_0 for S_2, then insert a fork block (F_0) and a join block (J_0) to execute D_0 and S_2 in parallel. Consequently, the second obfuscating transformation can be represented as $\langle O_D^S(\cdot, S_2), O_F^2(\cdot, D_0)\rangle$.

- Running $O_D^S(\cdot, S_2)$:

$$\mathbb{V} \leftarrow \mathbb{V} \cup \{D_0\},$$
$$\mathbb{E} \leftarrow \left(\mathbb{E} - \left\{(B_1, S_2)^T\right\}\right) \cup \left\{(B_1, D_0)^T, (D_0, S_2)\right\}.$$

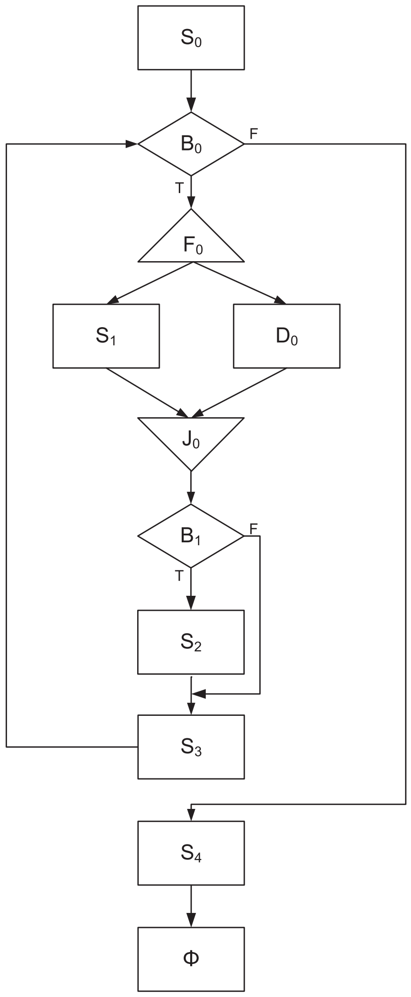
Again, we adopt *kidfunc()* as the dummy function in D_0, which is inserted as the true target of B_1. Now, D_0 and S_2 are in the loop led by B_1, so the range value of B_1 ($|range(\psi, B_1)|$) becomes 2 and $comp(\psi)$ can be derived as $|range(\psi, B_0)| + |range(\psi, B_1)| = 6 + 2 = 8$.

- Running $O_F^2(\cdot, D_0)$:

$$\mathbb{V} \leftarrow \mathbb{V} \cup \{F_0, J_0\},$$
$$\mathbb{E} \leftarrow \left(\mathbb{E} - \left\{(B_1, D_0)^{True}, (D_0, S_2), (S_2, S_3)\right\}\right)$$
$$\cup \left\{(F_0, D_0), (F_0, S_2), (J_0, S_3), (B_1, F_0)^{True}, (D_0, J_0), (S_2, J_0)\right\}.$$

$O_F^2(\cdot, D_0)$ makes D_0 and S_2 executed in parallel after F_0, as illustrated in Fig. 8. The insertion of F_0 contributes the parallelism such that $pcomp(\psi) = |range(\psi, F_0)| = 2$ and the total complexity of Program III is obtained as $|range(\psi, B_0)| + |range(\psi, B_1)| + |range(\psi, F_0)| = 8 + 4 + 2 = 14$.

By Eq. (1), the potency of $\mathcal{T}_2$ equals 14/6−1 = 4/3. The different potency values prove that the effect launched from



S_0	int num=4, tmp=7, sum;
	int PrimeOrNot=0;
	pthread_t kid;
	printf("Insert a number: \n");
	scanf("%d",&num);
	sum=0;
	tmp=2;

B_0	tmp <= num
F_0	pthread_create(&kid,NULL,kidfunc,NULL);
S_1	PrimeOrNot = _PrimeGen(tmp);
D_0	kidfunc();
J_0	pthread_join(kid,NULL);
B_1	PrimeOrNot == 1
S_2	printf("%6d",tmp);
S_3	sum += tmp;

S_4	printf("\n");
	return 0;

Fig. 7 – Program II: The obfuscated result of Program I after applying $\mathcal{T}_1 = \langle \mathbf{O}_D^S(\cdot, \mathbf{B}_1), \mathbf{O}_F^2(\cdot, \mathbf{S}_1)\rangle$.

a single obfuscating transformation may change with the different targets and the parameters of an obfuscating transformation. By comparing $\mathcal{T}_1$ and $\mathcal{T}_2$, we demonstrate that our evaluation method provides flexibility and distinguishability between different obfuscating transformations. We also conjecture that $\mathcal{T}_2$ provides more protection to ψ than $\mathcal{T}_1$ since $\mathcal{T}_2$ leads to a larger potency.

6.3. *Discussion*

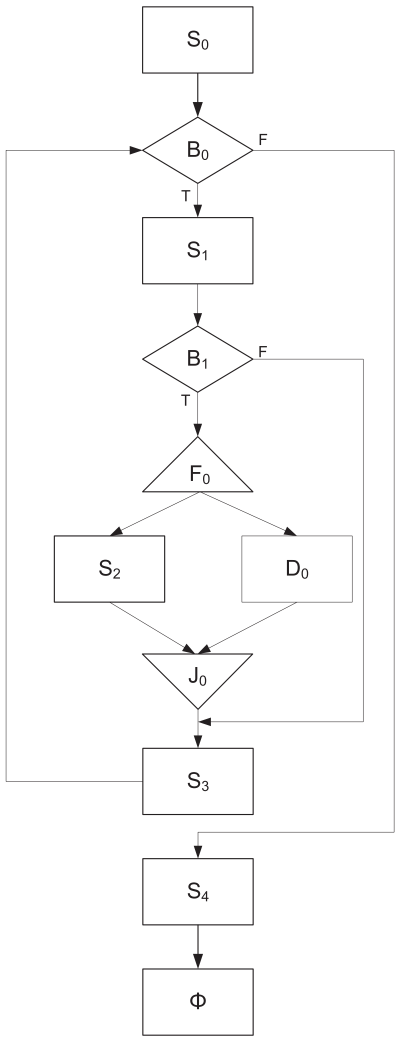
A coarse grained potency value is obtained if the dummy block with kidfunc() is treated as a single code block. If we further parse the dummy block D_0 into sub CFGs, we can obtain a finer grained result. First, *kidfunc()* is converted into ψ^K such that $\psi^K = (C^E, (\mathbb{V}, \mathbb{E}))$, where $C^E = S_0^K$, $\mathbb{V} = \{S_0^K, B_0^K, S_1^K, \phi\}$ and $\mathbb{E} = \{(S_0^K, B_0^K), (B_0^K, S_1^K)^T, (S_1^K, B_0^K), (B_0^K, \phi)^F\}$. The superscript K is used to emphasize that the code blocks are part of *kidfunc()* in D_0. Fig. 9 shows the CFG of ψ^K, of which the node count is 3.

Since there are 2 condition expressions in B_0^K, we consider the impact brought by these expressions, measure the SCOPE value contributed by B_0^K and obtain $|range(\psi^K, B_0^K)| = 2 \times (2 + 2 - 1) = 6$. Next, we integrate the complexity of ψ with that of ψ^K and gain $range(\psi, F_0) = \{S_1, S_0^K, S_1^K, B_0^K\}$, where S_0^K, S_1^K, B_0^K are code blocks in ψ^K.

Hence, $pcomp(\psi)$ becomes 4, and the range of B_0 increases such that $|range(\psi, B_0)| = 10$. The sequential complexity is derived as $|range(\psi, B_0)| + |range(\psi, B_1)| + |range(\psi^K, B_0^K)| = 10 + 1 + 6 = 17$. The finer grained complexity of Program II is then computed as $comp(\psi) = 17 + 4 = 21$ (assume that w_s and w_p in Eq. (2) are assigned equal weights). Consequently, we can get a finer grained potency $21/6 - 1 = 5/2$, which is larger than the coarse grained potency 5/6 obtained in Section 6.2. Such a comparison shows that the finer grained evaluation contains more information, and can be used to measure the effect caused by a newly inserted dummy code block in more detail.

7. Conclusion

In this paper we propose a framework toward evaluation of parallel control-flow obfuscating transformations. We define a parallel control-flow graph and design an atomic operator of inserting a fork based on the defined parallel CFG. This new operator complements the deficiency of existing atomic operators, which were designed to model sequential control-

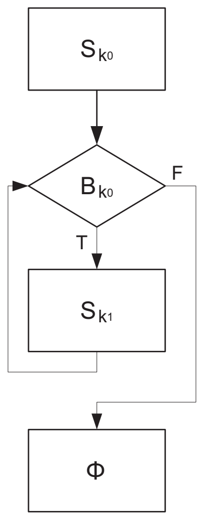


S_0	int num=4, tmp=7, sum;
	int PrimeOrNot=0;
	pthread_t kid;
	printf("Insert a number: \n");
	scanf("%d",&num);
	sum=0;
	tmp=2;

B_0	tmp <= num
F_0	pthread_create(&kid,NULL,kidfunc,NULL);
S_1	PrimeOrNot = _PrimeGen(tmp);
D_0	kidfunc();
J_0	pthread_join(kid,NULL);
B_1	PrimeOrNot == 1
S_2	printf("%6d",tmp);
S_3	sum += tmp;

S_4	printf("\n");
	return 0;

Fig. 8 – Program III: The obfuscated result of Program I after applying $\mathcal{T}_2 = \langle \mathbf{O}^{S}_{D}(\cdot,\mathbf{S_2}), \mathbf{O}^{2}_{F}(\cdot,\mathbf{D_0})\rangle$.



S_{k0}	int t1=*t;
B_{k0}	t1!=0& t1%10!=0
S_{k1}	t1--;

Fig. 9 – ψ_k : The CFG of the function *kidfunc*.

flow obfuscating transformations. Our new operator, along with the existing atomic operators, constitute together a set of building blocks to formalize both parallel and sequential control-flow obfuscating transformations. Moreover, we propose a new complexity measure which supports both sequential and parallel programs. The proposed measure considers the number of condition expressions in a branch that leads to a finer grained result. The measure can be used as the base of the potency metric to estimate the effect of obfuscation. We give examples to show that our method can even discriminate the effects caused by a single transformation when applying on different target code blocks. The examples imply that selections of target code blocks can be a factor toward protection of a program by an obfuscating transformation, in addition to the transformation algorithm itself. For this reason, we believe our framework can achieve stronger protection with the benefit from the target code blocks of an obfuscating transformation.

This paper presents a framework of evaluating parallel control-flow obfuscation. Currently, the framework is merely designed for measuring difficulty in reverse engineering through static analysis. We recognize that more work should be done for choosing between measuring results, human behaviors and endeavors required by dynamic analysis. We hope that we will be able to conjecture the effort and cost that an reverse engineer needs from the proposed measure, and then our framework will be helpful in examining the protection brought by obfuscation in a greater depth.

Acknowledgment

This effort was partially supported by Taiwan Information Security Center (TWISC), sponsored by National Science Council under the grants NSC100-2219-E-009-005, NSC-100-2218-E-009-003, and NSC- 100-2218-E-009-017.

REFERENCES

Anckaert B, Madou M, Sutter BD, Bus BD, Bosschere KD, Preneel B. Program obfuscation: a quantitative approach. In: Proceedings of the 3rd Workshop on Quality of protection (QoP'07); 2007. p. 15–20.

Barak B, Goldreich O, Impagliazzo R, Rudich S, Vadhan S, Yang K. On the (Im)possibility of obfuscating programs. In: Lecture Notes in Computer Science. Springer-Verlag; 2001. p. 1–18.

Ceccato M, Penta MD, Nagra J, Falcarin P, Ricca F, Torchiano M, Tonella P. Towards Experimental evaluation of code obfuscation techniques. In: Proceedings of the 4th Workshop on Quality of Protection; 2008. p. 39–46.

Ceccato M, Penta MD, Nagra J, Falcarin P, Ricca F, Torchiano M, Tonella P. The effectiveness of source code obfuscation: an experimental assessment. In: Proceedings of IEEE 17th International Conference on Program Comprehension (ICPC'09); 2009. p. 178–87.

Cheng J. Complexity metrics for distributed programs. In: Proceedings of Fourth International Symposium on Software Reliability engineering; 1993. p. 132–41.

Collberg C, Thomborson C, Low D. A Taxonomy of obfuscating transformations. Technical Report 148; Univ. Auckland; New Zealand; 1997.

Cota B, Fritz D, Sargent R. Control flow graphs as a representation language. In: Proceedings of Simulation Conference; 1994. p. 555–9.

Kazuhide F, Shinsaku K, Toshiaki T. An obfuscation scheme using affine transformation and its implementation. Transactions of Information Processing Society of Japan 2006;47(8):2556–70.

Lynn B, Prabhakaran M, Sahai A. Positive results and techniques for obfuscation. In: EUROCRYPT 04; 2004.

Madou M, Anckaert B, Bus BD, Bosschere KD, Cappaert J, Preneel B. On the effectiveness of source code transformations for binary obfuscation. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP06); 2006.

Majumdar A, Drape S, Thomborson C. Metrics-based evaluation of slicing obfuscations. In: 3rd International Symposium on Information Assurance and Security; 2007a. p. 472–7.

Majumdar A, Drape S, Thomborson C. Slicing obfuscations: design, correctness and evaluation. In: Proceedings of the 2007 ACM workshop on Digital Rights Management (DRM'07); 2007b. p. 70–81.

Naeem N, Batchelder M, Hendren L. Metrics for measuring the effectiveness of decompilers and obfuscators. In: Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07); 2007. p. 253–8.

Popov IV, Debray SK, Andrews GR. Binary obfuscation using signals. In: Proceedings of the 16th USENIX Security Symposium; 2007. p. 275–90.

Shatz SM. Towards complexity metrics for Ada tasking. IEEE Transactions on Software Engineering 1988;14(8): 1122–7.

Stotts PD, Cai ZN. Hierarchical graph models of concurrent CIM systems. In: Proceedings of IEEE Workshop on Languages for Automation: Symbiotic and Intelligent Robots; 1988. p. 100–5.

Tsai HY, Huang YL, Wagner D. A graph approach to quantitative analysis of control-flow obfuscating transformations. IEEE Transactions on Information Forensics and Security 2009;4(2): 257–67.

Udupa S, Debray S, Madou M. Deobfuscation: reverse engineering obfuscated code. In: Proceedings of 12th working Conference on reverse engineering (WCRE 2005); 2005.

Wang C, Davidson J, Hill J, Knight J. Protection of software-based Survivability Mechanisms. Foundations of Intrusion Tolerant Systems; 2003. 273–82.

Zuse H. Software complexity: measures and methods. Hawthorne, NJ, USA: 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 2005, and 2007 respectively. She received the Ph.D. degree in Electrical Control Engineering from the National Chiao-Tung University, Taiwan in 2011. Her research interests include2 evaluation of protection techniques, risk assessment of networks, and design of security metrics. Ms. Tsai has been 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. She has been a member of Phi Tau Phi Society since 1995. She is now an assistant professor in the Department of Electrical Engineering of National Chiao-Tung University. Her research interests include wireless security, secure testbed design, embedded software, embedded operating systems, risk assessment, secure payment systems, VoIP, and QoS.