Program obfuscation as obstruction of program static analysis

Kirill S. Ivanov, Vladimir A. Zakharov

Abstract. By obfuscation we mean any efficient semantic-preserving transformation of computer programs aimed at bringing a program into such a form, which impedes the understanding of its algorithm and data structures or prevents the extracting of some valuable information from the plaintext of a program. Some results obtained recently testify that it is very much uneasy to develop a provably secure program obfuscator unless a capability of an adversary is restricted weakly. But in many practical cases when program re-engineering activity is restricted only to the application of available static analysis algorithms an acceptable security of obfuscation can be achieved by guaranteeing the obfuscated programs to obstruct completely some definite static analysis technique. This approach to designing program obfuscators is commonly used now though no rigorous study of its efficiency and security is made so far. In this paper we give a preliminary framework for studying formally program obfuscators as tools for the obstruction of program static analysis. We present a criterion which provides a way of estimating the actual efficiency of static analysis obstruction and demonstrate with example of two typical static analysis algorithms (Andersen's points-to analysis algorithm and Binkley-Horwitz-Reps slicing algorithm) how to use our approach in practice.

1. Introduction

Protection of software against intelligent tampering and unauthorized purposeful modifications is one of the central issues in computer security. An advent of global computer networks has given rise to new computational environments which provide remote execution, distributed computing, code mobility etc. Although new models of computation bring great flexibility and new promises to the world of computing they are accompanied with a set of new problems that were not present when computation was carried out for the most part on local stand-alone machines.

One of the most difficult security problem is that of protecting a code from an untrustworthy environment which is known as *malicious host problem*. One faces the challenge of the malicious host problem every time when downloads or installs a code on a host computer which is shared by a second party. In this case a client program becomes a subject of various threats, such as

- *software piracy*, the illegal copying and resale of applications;
- *malicious reverse engineering*, the illegal extraction of valuable pieces of code for the purpose of unauthorized using of them in other applications;
- *program tampering*, the illegal extraction of valuable information from a code for the purpose of unauthorized using of the code.

Unlike many similar problems (say, malicious code problem) the malicious host problem has not been studied very much intensively and rigorously. Various methods and tools are widely used for the purpose of software protection, including sophisticated security policies, network filters, cryptosystems, tamper-resistant hardware, etc. [1, 12, 13, 19]. But no matter how powerful these techniques may be, they don't cover the case when an adversary, having in mind to make an illegal modification of a program or to gain some valuable knowledge about algorithms or data structures, gets an access to the plaintext of a program. The current trends in software engineering and communication technology make it common to distribute software in such a form that hold most of information presented in the program source code. This increases drastically the risk of reverse engineering attacks aimed at extracting secret information from a program. An important example is Java bytecode. Java applications are distributed as Java class files, hardware-independent virtual machine codes that retains almost all information of the original Java source. The customary cryptographic tools can be very effective for protecting programs from illegal usage at the stage of their distribution. But when a program is decrypted, it becomes extremely vulnerable to software pirates seeking for private information (passwords, data-keys, etc.) or valuable pieces of code to incorporate them in their own applications.

In these cases the only way to prevent such malicious activity is to convert a program into some tamper-resistant form, which has the property that understanding and making purposeful modifications to it are rendered difficult while its original functionality and efficiency are preserved. Program transformations of this kind are called *obfuscating transformations*.

Obfuscation attempts to transform a program into an equivalent one that is harder for understanding and reverse-engineering. The concept of program obfuscation was introduced in [8]. In this paper an obfuscator is defined informally as any efficient probabilistic compiler $\mathcal{O}$ which transforms any source program π into an equivalent program $\mathcal{O}(\pi)$ which is substantially less intelligible (readable) than π. In succeeding papers [9, 10] this concept has been specified in more details and the program obfuscation problem has been set up as follows.

> Given a set of programs $\mathcal{P}$, $\mathcal{P} \subseteq \mathcal{U}$, find a transformation $\mathcal{O} : \mathcal{P} \to \mathcal{U}$ such that for every program π from $\mathcal{P}$ an obfuscated program $\pi' = \mathcal{O}(\pi)$ should satisfy the following requirements.

semantic-preservation: π' has the same observable behavior as π;

obscurity: understanding and reverse engineering π' is strictly more time-consuming than understanding and reverse engineering π;

resilience: it is either difficult to construct an automatic tool to undo the obfuscation transformation, or executing such a tool is extremely time-consuming;

stealth: the statistical properties of π' are similar to those of π;

cost: execution time/space penalty incurred by the obfuscation of π are small.

The key issue in this definition is an obscurity of program obfuscation. Unlike other requirements this item does not admit a uniform indubitable formalization since it involves such loose concepts as "understanding", "semantic interpretation", etc. Nevertheless in [3, 14, 27] serious efforts were made to find some suitable formalism for capturing the obscurity requirement. As early as 1998 J.MacDonald [22] noticed that to protect a program "we are in search of security, however, and not obscurity, so a formal proof of security is a necessity".

To formalize the obscurity requirement Barak, Goldreich, et al. introduced in [3] the concept of "perfect" obfuscation. Intuitively, a program obfuscator $\mathcal{O}$ is called *perfect* if it transforms any program π into a "virtual black box" $\mathcal{O}(\pi)$ in the sense that anything one can efficiently compute given $\mathcal{O}(\pi)$, one should be able to efficiently compute given just oracle access to π. The main result of [3] implies that perfect obfuscation is impossible. This is proved by constructing a family $\mathcal{F}$ of functions and a predicate $P : \mathcal{F} \to \{0, 1\}$ such that, given any program that computes a function $f \in \mathcal{F}$, the value $P(f)$ can be efficiently computed, whereas no efficient algorithm, being given only oracle access to a randomly selected $f \in \mathcal{F}$, can compute $P(f)$ much better than by random guessing.

An alternative approach to the development of the rigorous concept of obfuscation security was initiated in [27]. In this paper a formal information-theoretic definition of obfuscation security was introduced. Informally, $\mathcal{O}$ obfuscates securely a property P of programs from some ensemble $\mathcal{H}$ if any adversary A, being granted anaccess to programs from $\mathcal{O}(\mathcal{H})$, can extract in polynomial time only a "negligibly small" amount of information on P. This definition was also provided with novel obfuscation technique which takes advantage of cryptographic primitives (one-way functions, hard-core predicates). To show that the new formalization of obfuscation security enables one to design provably secure program obfuscators one meaningful example was considered as a case study. In this example a class of programs that implement a simple fixed password identification scheme is taken for $\mathcal{H}$, and the property of a program to execute actual password checking is considered as a secret property P . A program obfuscator

$\mathcal{O}$ is designed and, assuming that one-way permutations exist, it is proved that $\mathcal{O}$ securely obfuscates P. Moreover, close relationships between obfuscation problem and basic cryptographic primitives was established by demonstrating that a secure obfuscation of P implies the existence of one-way functions.

But the mainstream of research in program obfuscation has been focused on the development of obfuscation techniques which admit a straightforward application to the solution of malicious host problem without giving strong evidence of their security. A taxonomy of basic obfuscating transformations was presented in [8]. It includes

lexical transformations: scrambling of identifier, deleting comments, etc.;

control transformations: inserting dead pieces of code and faked branchings, unfolding of loops, splitting and merging control flow structures etc.;

data transformations: variable splitting, data encoding, converting static data into dynamic, etc.

The most important concept introduced in [8] is that of *opaque predicate*. A predicate R is called opaque if its outcome is known at obfuscation time but is difficult for the deobfuscator to deduce. Given such opaque predicates it is possible to construct obfuscating transformations that entangle control flow of a program so that no adversary could understand the intended meaning of the program components. It looks like the very essence of obfuscation is concentrated in this notion and the development of secure opaque predicates may be regarded as a prime task of program obfuscation.

The main difficulty in solving this task is in providing security of opaque predicates. But if an adversary is assumed to be restricted in her/his potency and only currently available automatic tools are allowable for deobfuscation then an obscurity of an opaque predicate R can be achieved by obstructing relevant program static analysis techniques. It is well known (see [17, 20, 26]) that precise static analysis of programs containing complex data structures and pointers is either undecidable or extremely time consuming. Therefore one could obstruct a static analysis toolset by using for obfuscation such an opaque predicate whose precise analysis by means of the selected toolset is a hard problem.

This approach was implemented in many papers. In [9, 11] it was shown how strong opaque predicates can be constructed based on intractability of alias analysis. The basic idea is to extend the program to be obfuscated with a code that builds a set of complex dynamic structures. The introduced code will occasionally update the structures (modifying pointers, splitting and merging fields, etc.) while maintaining certain invariants. These invariants are then used to manufacture opaque predicates as needed. In [6, 28] it is demonstrated that 3-SAT problem which is known to be NP-complete can be simulated by means

of general pointers. This makes it possible to design opaque predicates whose precise analysis is an NP-complete problem. A more sophisticated using of function pointers is used in [24] for the same purpose. In [7] it was shown that any PSPACE-complete problem can be used as the basis for manufacturing opaque predicate and we could restrict ourselves to scalar boolean variables only.

The principal drawback of all these techniques is that none of them has a formal basis for making claims about the difficulty of understanding an obfuscated program. Therefore it is hardly possible to estimate to what extent such methods serve the purpose — only the reference to the intractability of some static analysis problems or to the hardness of combinatoric problem embedded into obfuscated programs is not sufficient. Some highly speculative measures for potency and resilience of obfuscating transformations were introduced in [6, 8, 30] for the purpose of estimating the potency of obfuscating transformations. Usually, given a measure of program complexity E, a potency of obfuscating transformation $\mathcal{O}$ in relation to obfuscated program π is defined as

$$\Pi(\mathcal{O}, \pi) = \frac{E(\mathcal{O}(\pi))}{E(\pi)} - 1.$$

The problem of measuring program complexity is quite long-term. In the frame of software theory lot of different measures were created (see [6] for survey). But all these measures characterize only static issues of program (e.g. length of program, number of variables, cyclomatic number of control flow graph, etc) and none of them gives an acceptable fit to our purpose.

In this paper we introduce a framework for the rigorous study of program obfuscators as tools for the obstruction of program static analysis. Our main idea is as follows. A program static analysis tool A computes a mapping $A : \mathcal{U} \to W$, where $\mathcal{U}$ is a set of programs under consideration and W is a set of possible outcomes of static analysis. For example, when static analysis is applied for detecting useless variables a powerset $\mathcal{P}(Var_\pi)$, where Var_π is the set of all variables of a program π, stands for W. As a rule, a set W of possible outcomes constitutes a lattice which has the minimal element $\perp$. This distinguished element may be interpreted as a lack of any significant information about the program property analyzed by means of A. In the case of useless variables detection $\perp = \emptyset$ which means that static analyzer A can not detect any useless variable.

Definition. Given a program obfuscator $\mathcal{O}$ and a program static analyzer A we say that $\mathcal{O}$ is *impervious* to A if $A(\mathcal{O}(\pi)) = \perp$ holds for any program π.

Thus, a program obfuscator $\mathcal{O}$ impervious to static analyzer A makes this toolset unsuitable for the purpose of program decompilation. Imperviousness criterion has several advantages over other measures for the security/obscurity

of obfuscating transformations. Unlike complexity-theoretic [3, 14] and information-theoretic [27] criteria it may be used not only in theory but also in practice for demonstrating resilience of certain obfuscating transformations against possible reverse engineering attacks. Imperviousness criterion defined thus does not appeal to the worst-case complexity of static analysis (cf. [7, 8, 9, 10, 11, 24, 28, 30]) and, hence, it could be used to guarantee absolute resilience of obfuscating transformations against some specific decompilation tools. Finally, given a specific static analysis algorithm A and an obfuscation transformation $\mathcal{O}$ one could prove formally that $\mathcal{O}$ is impervious to A. The aim of this paper is to demonstrate that the latter is possible at least for some simple program analysis algorithms. We take two typical static analysis algorithms (Andersen's points-to analysis algorithm A_1 and Binkley-Horwitz-Reps slicing algorithm A_2) as examples. For each of these algorithms A_i, $i = 1, 2$, we develop a corresponding obfuscating transformation $\mathcal{O}_i$ and prove that $\mathcal{O}_i$ is impervious to A_i.

The paper is organized as follows. In Section 2 we discuss briefly the problems of pointer analysis and slicing for computer programs. We introduce also the model of programs these algorithms deal with. In Section 3 we consider in more details a flow-insensitive points-to analysis algorithm A_1 presented in [2], introduce our obfuscating transformation $\mathcal{O}_1$ aimed at obstructing A_1, prove its imperviousness to A_1, and estimate the cost of obfuscation. In Section 4 the same investigation is applied to a slicing algorithm A_2 presented in [18]. Finally, in Section 5 we discuss further lines of research.

2. Program static analysis

Program static analysis is a branch in formal methods which incorporate a variety of techniques intended for deducing the safe information about program behavior immediately from the text of a program. Static analysis holds an intermediate position between complete formal verification of programs and program testing. Static analysis algorithms are used mostly for revealing some specific semantic properties of programs (like dead code detection, constant propagation, etc.) which may be helpful for program optimization, understanding, re-using, etc. The efficiency of such algorithms is characterized by complexity and accuracy. Therefore to obstruct static analysis one has to bring a program into such a form which is either intractable for the current static analysis algorithms or drastically degrade their accuracy. Since many static analysis algorithms operate in polynomial time, the only way to obstruct them is to reduce the precision of their outcome to the minimal possible level.

In this article we will consider two typical static analysis algorithms. One of these algorithms is intended for detecting aliasing. Two variables x and y are said to be *aliasing* if they could refer to the same memory location during a program execution. Alias analysis is necessary for revealing implicit updating

of variables. It is well known that aliasing problem is undecidable in general (see [26]) and remains hard even for simple programs (see [5, 17]). Nevertheless in [2, 23] (see also [4, 15])it was shown that some reasonable approximation for pointer aliasing can be computed in polynomial time. Each of these algorithms associates a pointer p with a set of variables W_p^c and guarantees that no variables beyond W_p^c can be alias of p. The precision of an alias analysis algorithm is characterized by the ratio $\frac{\#(W_p^c)}{\#(W_p^a)}$, where W_p^a is the set of actual aliases of a pointer p. Clearly, the least possible precision is attained when W_p^c coincides with the set of all variables declared in the program. In this case alias analysis by means of fast algorithms from [2, 23] are completely obstructed. In the next section we will demonstrate how to transform a program into such a form which guarantees a complete obstruction of fast alias analysis. According to the definition above this program transformation is impervious to the fast alias analysis. For the sake of simplicity we restrict ourselves with flow-insensitive and context-insensitive variants of alias analysis algorithms.

Another type of static analysis algorithms considered in this paper are static slicing analysis algorithms. Program slicing is a viable method to restrict the focus of a task to specific sub-components of a program. A slice [29] is an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point. The applications of slicing technique include debugging, testing, program comprehension, restructuring, downsizing, and parallelization. The most advanced algorithms (see, e.g. [18]) compute static slices as backward traversals of a Program Dependence Graph (PDG) [25], a program representation where nodes represent statements and predicates while edges carry information about control and data dependencies. A PDG based algorithm considers slicing criteria of the type $\langle p, V\rangle$, where p is a program point and V is a set of variables referenced at p. A slice with respect to such a slicing criterion consists of the set of nodes that directly or indirectly affect the computation of of the variables in V at node p.

Since slices are very much helpful for revealing hidden dependencies between data structures and program components, program obfuscation has to be resilient against slicing algorithms.The efficiency of a slicing algorithm C is characterized by its time complexity and by the ratio $\frac{\#(S_{p,V}^C)}{\#(S_{p,V}^a)}$, where $S_{p,V}^C$ is the set of program points included into the slice computed by the algorithm with respect to a slice criterion $\langle p, V\rangle$, and $S_{p,V}^a$ is the minimal set of program nodes that are necessary to be kept intact for computing correctly the values variables from V at point p. The worst possible solution to static slicing problem is given by the trivial slicing algorithm which outputs as $S_{p,V}^c$ the set $S_{p,V}^w$ of all program points that precede p in a Control Flow Graph. Thus, according to the definition above program obfuscation is said to be impervious to a slicing

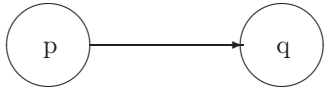
algorithm C if for every slicing criterion $\langle p, V\rangle$ an algorithm C after being applied to an obfuscated program $\mathcal{O}(\pi)$ outputs the worst possible slice $S_{p,V}^w$ as the result. In this case static slicing algorithm is completely obstructed. In the next section we will demonstrate how to transform a program into such a form which guarantees a complete obstruction of static slicing algorithm introduced in [18].

For the sake of clarity we restrict ourselves with considering programming language whose syntax includes only simple data types, assignment statements, **if-then-else** statements, **while-do** statements and function calls.

3. Obstructing alias (points - to) analysis

3.1. Points-to analysis algorithms

Given a program π a points-to analysis algorithm C computes a function $f_{C,\pi} : Var \to 2^{Var}$ such that for every variable x the set $f_{C,\pi}(x)$ includes all variables x may refer to along some run of π. Usually the function $f_{C,\pi}$ is presented as a directed graph whose nodes are associated with variables or sets of variables. An arc from a node p to a node q means that q is included in $f_{C,\pi}(x)$.



Since the problem of detection given a program π and a pair of variables p, q whether π admits a computation such that p refers to q is undecidable, any function $f_{C,\pi} : Var \to 2^{Var}$ computed by a points-to analysis algorithm only approximates true points-to relation between program variables.

We will consider three points-to analysis algorithms presented in [23], [16], and [2]. The most fast algorithm is that by Steensgaard [23] (in what follows this algorithm will be identified as S-algorithm), and the most precise is that by Andersen [2] (A-algorithm). The points-to analysis algorithm presented by Shapiro and Horwitz [16] (SH-algorithm) has a parameter k. This parameter allows one to tune speed and precision of the algorithm. If $k = 1$ then it gives the same result as that of Steensgaard's algorithm, and if $k = n$ (where n is the number of variables) then it's result is the same as that of Andersen's algorithm.

To explain in more details the difference between these algorithms let us consider the following program:

a = &b;

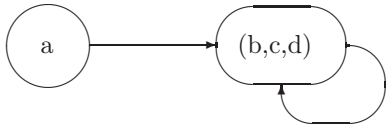
a = &d;

a = &c;

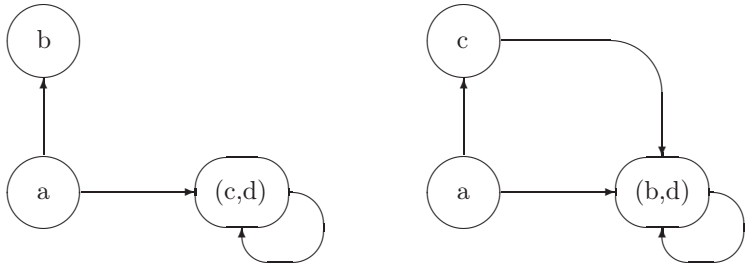
c = &d;

The results of each algorithm are as follows:

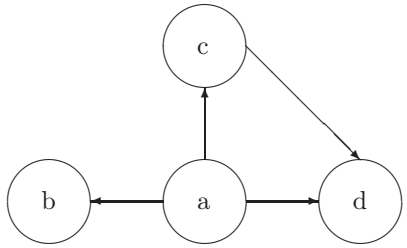
1. Steensgaard's algorithm (Shapiro and Horwitz with $k = 1$)



2. Shapiro and Horwitz algorithm with $k = 2$



3. Shapiro and Horwitz algorithm with $k = 3$ (Andersen's algorithm)



3.2. Equivalent transformations

Any insertion of the assignment statement in some program point may change the value of at least one variable and affect program runs. To preserve program behavior we will use only those transformation that restore or keep the value

of the affected variable until it's next use in predicate, calling of a function or in a right-hand side of some assignment statement. Examples of equivalent transformations are presented below.

1. *Use of opaque predicates.* A predicate P is called *opaque* predicate if it's value is constant and it is known before the compilation but is hard for computing by means of static analysis algorithms. Instructions of the inserted block that affect the value of p can be hidden in a branch of conditional statement with a predicate $P \equiv 0$. For example:

```
...
p = &b;
...
//beginning of a block
...
if(P) // P ≡ 0
   p = q; // hiding obfuscation instruction
...
a = *p + 3c;
...
//end of a block
```

2. The simplest way to preserve the values of selected variables is to use extra variable. In what follows it will be demonstrated that extra variable won't affect anything in the obfuscation process in this case. Example:

```
...
p = &b;
...
// inserting a block
...
v = p; //storing a value
p = q;
...
p = v; //restoring a value
a = *p + 3c;
...
```

3.3. Obfuscation algorithm $\mathcal{O}_1$

Since Andersen's algorithm [2] is the most precise points-to analysis algorithm, it is suffice to show how to obstruct this static analysis technique.

The main weakness of points-to analysis algorithms under consideration lies in their flow-insensitivity. If some fragment contains a pair of instructions

```
...
p = &a;
...
q = &a;
...
```

then Andersen's algorithm characterizes variables p and q as an aliased pair of pointers in relation to value a independently of points of their occurrence in a program. One should also notice that instructions like $p = q$ expand the set of possible values of p with the set of q ($X_p = X_p \cup X_q$, where X_i is the set of possible values for pointers i).

The main idea of our obfuscation technique is based on inserting additional assignment and de-reference statements and using additional pointer variables aiming at making all pointers of the same reference level aliased (their set of possible values will be equal). Since now by the *set of pointer p* we will mean a set of possible values of a pointer p at the end of the points-to analysis. The obfuscation algorithm is as follows:

1. *Pointer Analysis.*
   - collecting information about pointers, determine the maximum pointing level (mark it as max_level);
   - collecting the set V of all variables (we will denote by V_i the subset of V containing pointers of level i).
   - extracting instructions $p = q$; and coping them into the set W in the form of $p \to q$;
   - extracting de-reference instructions $p = \&a$; and coping a into the set S_p.

   The sets W and S_p will be used later to avoid superfluous insertions of statements.
2. *Optimization of the Set W.* Since the set W can contain cycles (e.g. it can contain both pairs $p \to q$ and $q \to p$), it requires optimization. The optimization procedure is as follows:
   - build a directed graph whose arcs are all pairs from W;
   - delete nodes that have no in-going arcs;
   - mark all nodes and arcs that form cycles in the graph;
   - delete all marked nodes that has no marked in-going and out-going arcs. If there are no such nodes in a cycle then select randomly any node in the cycle and delete all other nodes;

   - delete all nodes that has no in-going edges.

   We deleted thus all pointers that are necessary for the obfuscation algorithm. The obfuscation is not necessary for pointers remaining in W since the instruction $p = q$; will ensure that the set of p is larger than the set of q. And the set of q will cover the entire level.
3. *Preparing Additional Variables.*

   At this step the set of additional pointer variables is defined. Let $k \geq 1$ be an input parameter indicating the number of additional pointer per reference level. So we have $k * max\_level$ additional pointers. Denote by P_i is the set of such variables on the i-th level, $|P_i| = k$, $i = \overline{1, max\_level}$, $P_0 = \emptyset$.
4. *Building Lists of Additional Assignments.*

   We describe this step for the i-th level, $i = \overline{1, max\_level}$.

   (a) Denote by $(V_{i-1} \cup P_{i-1})_p$ the set of variables associated with a pointer p. This set should satisfy the following requirements:
   - For any pair $s, p \in P_i$

     $$\begin{array}{rcl} (V_{i-1} \cup P_{i-1})_s & \subset & (V_{i-1} \cup P_{i-1}), \\ (V_{i-1} \cup P_{i-1})_p & \subset & (V_{i-1} \cup P_{i-1}). \end{array}$$

   - If $s \neq p$ then $(V_{i-1} \cup P_{i-1})_s \cap (V_{i-1} \cup P_{i-1})_p = \emptyset$,
   - $\bigcup\limits_{p \in P_i} (V_{i-1} \cup P_{i-1})_p = (V_{i-1} \cup P_{i-1})$.

   For each $p \in P_i$ and $q \in (V_{i-1} \cup P_{i-1})_p$ form a de-reference assignment $p = \&q$; and add it to the list $List_1$. Thus, additional pointers have the set of values from the lower level.

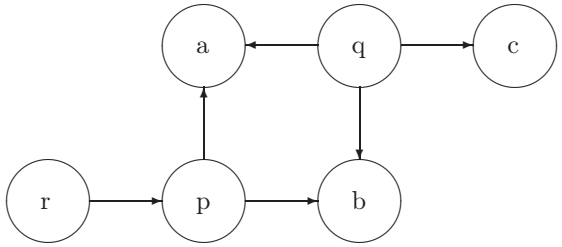
   (b) For each $v \in V_i \cap (V \setminus W)$ and for each $p \in P_i$ such that $(V_{i-1} \cup P_{i-1})_p \nsubseteq S_v$ form an assignment statement $v = p$; and add it to the list $List_2$. Thus, every pointer at the level i will gain a set of possible values equal to the set of variables from the lower level. Now expand the set for additional variables. For each $p \in P_i$ and any randomly selected $v \in V_i$ form an assignment statement $p = v$; and add it to the list $List_1$.
5. *Inserting additional statements.*

   Add statements from the lists A and I into the program code. Statements from A can be inserted evenly. Statements from I must be inserted in according to the rule specified in Section 3.2).

An example below demonstrates how this algorithm operates. Let us consider the following program π.

```
int a, b, c *p, *q, **r;
  p = &b;
  q = &c;
  r = &p;
  a = **r + *q;
  q = p;
  p = &a;
```

By applying Andersen's points-to analysis algorithm we obtain the reference graph for π:



After applying our obfuscation algorithm for $k = 2$ we obtain the following program $\mathcal{O}(\pi)$:

Step 1&2: $V_0 = \{a, b, c\}, V_1 = \{p, q\}, V_2 = \{r\}, S_p = \{b, a\}, S_q = \{c\}, S_r = \{r\}$

Step 3: $P_0 = \emptyset, P_1 = \{s_{11}, s_{12}\}, P_1 = \{s_{21}, s_{22}\}$

Step 4: $A = \{$"s_{11} = &a;", "s_{11} = &b;", "s_{12} = &c;", "s_{21} = &p;", "s_{21} = &q;", "s_{22} = &s_{12};", "s_{22} = &s_{11};", "$s_{11} = p$;", "$s_{12} = q$;", "$s_{22} = r$;", "$s_{21} = r$;"$\}, I = \{$"$r = s_{21}$;", "$r = s_{22}$;", "$p = s_{12}$;", "$p = s_{11}$;"$\}$

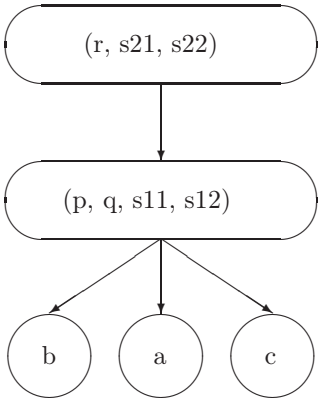
Step 5: Program $\mathcal{O}(\pi)$ (possible form):

```
int a,b, c,*p, *q, **r;
int *s11, *s12, **s21, **s22;
  s11 = &a;
  p = s11;
  s21 = &p;
  r = s21;
  s22 = r;
  p = &b;
  s11 = &b;
  s22 = &s11;
  q = &c;
  s21 = r;
  s12 = &c;
  if(0)  p = s12;
```



```
r = &p;
s22 =&s12;
a = **r + *q;
r = s22;
s11 = p;
q = p;
s21 = &q;
p = &a;
s12 = q;
```

Andersen's algorithm graph for $\mathcal{O}(\pi)$:



3.4. Correctness, complexity and overheads of the obfuscation $\mathcal{O}_1$

Transformations specified above preserve functions computed by programs, i.e. they are semantic-preserving (equivalent) transformation.

Claim 1. Let π be an arbitrary program and $\mathcal{O}_1(\pi)$ be a result of transformation of π as described in Section 3.3. Then $\mathcal{O}_1(\pi)$ computes the same function as π.

The obfuscating transformation $\mathcal{O}_1$ obstructs flow-insensitive points-to analysis algorithms by hiding differences between pointer variables and making thus each variable to be an alias of any other variable at the same level of the reference graph: for every obfuscated program $\mathcal{O}_1(\pi)$ Andersen's points-to analysis algorithm builds only the trivial reference graph as depicted in Section 3.3. Thus, we arrive at:

Claim 2. The obfuscating transformation $\mathcal{O}_1$ is impervious to flow-insensitive variant of points-to analysis algorithms (A-algorithm, S-algorithm and SH-algorithm).

Now we are to find upper bound of the obfuscating transformation complexity. Assume that a program π contains n statements and m variables. The first step (Pointer Analysis) requires $O(n)$ operations. The next step (Optimization of the Set W) takes about $O(m^2)$ operations on the reference graph. The maximum number of additional variables is $k*max\_level$, where k is a parameter used in SH-algorithm. So on the step of Preparing Additional Variables the algorithm builds $O(max\_level * k * m^2)$ additional instructions. The insertion of new instructions into the code takes $O(m * n)$ operations. So the total complexity of $\mathcal{O}_1$ is $O(max\_level * k * m^2 + nm)$.

Claim 3. Time complexity of obfuscating transformation $\mathcal{O}_1$ is $O(|\pi|^3)$.

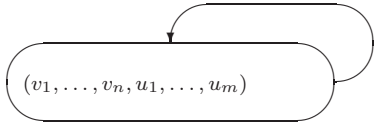
The overheads of the obfuscating transformation $\mathcal{O}_1$ can be characterized in terms of the number of additional instructions.

Claim 4. For any program π

$$|\mathcal{O}_1(\pi)| - |\pi| = O(m * k * max\_level)$$

If a program π is such that $n \gg m$ (which seems reasonable for most programs appeared in practice) then $\frac{|\mathcal{O}_1(\pi)|}{|\pi|} \approx 1$ and the obfuscating transformation $\mathcal{O}_1$ can be accepted as *cheap* according to the Taxonomy of Obfuscating Transformations [8].

The scripting languages appear to be one of the developing directions in the programming. The distinguished feature of scripts is that they are formally typeless. So our obfuscating transformation can be easily adopted to scripting languages. We have to replace all the occurrences of P_i with P, V_i with V and so on. Finally if $(v_1, \ldots, v_n)$ is the set of variables and $(u_1, \ldots, u_m)$ is the set of additional pointers, the analysis graph for the program will be as follows:



4. Obstructing program slicing

Any static slicing algorithm based on Program Dependence Graphs (PDG) begins to work with building PDG and the size of a slice obtained is determined by the size of PDG. PDG presents both control and data dependencies between variable. We say that a variable x at a program point p depends (semantically) on a variable y at a program point q if some spontaneous change of a value of y may affect either a value of x or an access to the point p via q by some execution

of a program. In general, dependency relation is intractable but admits safe approximation which can be computed efficiently relying only on the syntactic structure of a program: a program point p depends (syntactically) on a point q if

- some variable y is defined at q (e.g. y is in the left-hand side of an assignment statement), used at p (e.g. y occurs in the right-hand side of an assignment statement), and there is a syntactically admissible path from q to p in the control-flow graph of the program such that no other definitions of y occurs along this path;

or

- y is defined at q and used as an argument of a predicate at a branching point p and there is a syntactically admissible path from q to p such that no other definitions of y occurs along this path.

PDG represents all syntactic dependencies between program points.

4.1. Introducing extra dependencies

To obstruct static slicing algorithm we introduce some additional syntactic dependencies so that any point p in a program becomes dependent from *all* points q preceding p in the control flow graph of the program. In this case any slice will cover all such points giving thus no useful information about control and data dependencies in an obfuscated program. To this end we introduce a series of specific equivalent transformations.

T_1 Given an assignment statement S of the form $y = f(\vec{x})$ and an arithmetic expression $g(z)$ such that $g(z) \equiv c$ we may induce a dependency relation between y and an arbitrary variable z by replacing the above statement by $y = f(\vec{x}) + g(z) - c$. We will denote this transformation by $T_1^z(S)$.

T_2 The same effect can be achieved by replacing any occurrence of a variable x in a right-hand side of an assignment by the expression $\frac{x+z}{2} - \frac{x-z}{2}$. We will denote this transformation by $T_2^z(S)$.

T_3 Boolean identities also provide a vast possibility for inserting new dependencies:

1. $P = (R \wedge P) \vee P$;
2. $P = (R \wedge P) \vee (\neg R \wedge P)$.

This transformation will be denoted by $T_3^R(P)$.

4.2. Obfuscation algorithm

The obfuscation algorithm $\mathcal{O}_2$ presented in this section when being applied to a statement S transforms it given a list of variables L to an equivalent statement $\mathcal{O}_2(S, List)$ and builds a new list of variables $List(S, L)$ according to the following rules.

1. If a statement S is a sequential composition of statements S_1 and S_2 (i.e. $S = S_1;\ S_2$) then $\mathcal{O}_2(S, L) = \mathcal{O}_2(S_1, L);\ \mathcal{O}_2(S_2, L')$ and $List(S, L) = List(S_2, L')$, where $L' = List(S_1, L)$.

2. If S is an assignment statement of the form $x := t(y_1, \ldots, y_n)$ and $L \setminus \{y_1, \ldots, y_n\} = \{z_1, \ldots, z_k\}$ then $\mathcal{O}_2(S, L)$ is either $T^{z_1,\ldots,z_k}(S)_1$ or $T^{z_1,\ldots,z_k}(S)_2$, and $List(S, L) = \{x\}$.

3. If S is a conditional branching statement of the form **if** P **then** S_1 **else** S_2 **fi**, where P is a predicate whose arguments are variables $y_1, \ldots, y_n$, then $\mathcal{O}_2(S, L)$ is a statement of the form **if** $T_3^R(P)$ **then** $\mathcal{O}_2(S_1, \emptyset)$ **else** $\mathcal{O}_2(S_2, \emptyset)$ **fi**, where R is an arbitrary predicate whose arguments are all variables from the list $L \setminus \{y_1, \ldots, y_n\}$, and $List(S, L) = List(S_1, \emptyset) \cup List(S_2, \emptyset)$.

4. If S is an iteration of the form **while** P **do** S **od**, where P is a predicate whose arguments are variables $y_1, \ldots, y_n$, then $\mathcal{O}_2(S, L)$ is a statement of the form **while** $T_3^R(P)$ **do** $\mathcal{O}_2(S_1, \emptyset)$ **od**, where R is an arbitrary predicate whose arguments are all variables from the list $L \setminus \{y_1, \ldots, y_n\}$, and $List(S, L) = List(S_1, \emptyset)$.

Given a program π an obfuscation algorithm $\mathcal{O}_2$ begins to operate with an empty list of variables and results with a program $\mathcal{O}_2(\pi, \emptyset)$.

4.3. Correctness, complexity and overheads of the obfuscation $\mathcal{O}_2$

Since all transformations presented in Section 4.1 are semantic-preserving, the transformation $\mathcal{O}_2$ also preserves the function computed by a source program.

Claim 5. Let π be an arbitrary program and $\mathcal{O}_2(\pi, \emptyset)$ be a result of transformation of π as described in Section 4.2. Then $\mathcal{O}_1(\pi)$ computes the same function as π.

It is easy to demonstrate that the extra dependencies introduced by $\mathcal{O}_2$ make each statement S in a program $\mathcal{O}_2(\pi)$ dependent on all statements that precede S in the control flow graph of $\mathcal{O}_2(\pi)$. In this case any static slicing technique based on PDG outputs the worst possible result — for every slicing criteria $\langle p, x\rangle$, where p is a program point and x is a variable defined at p the corresponding static slice will include all statements that precede p in the Control Flow Graph of a program under consideration. Thus, we arrive at:

Claim 6. The obfuscating transformation $\mathcal{O}_2$ is impervious to any PDG-based static slicing technique.

Each transformation T_1, T_2, T_3 takes time $O(m)$, where m is a number of new variables inserted into a statement or a predicate, and each basic statement or a predicate is modified along the application of $\mathcal{O}_2$ at most once. Thus, we arrive at:

Claim 3. Time complexity of obfuscating transformation $\mathcal{O}_2$ is $O(|\pi|^2)$.

The obfuscating transformation $\mathcal{O}_2$ does not introduce new statements but it could extend some statements by inserting extra variables. In the worst case all variables of a program could be inserted into a statement.

Claim 4. For any program π

$$\frac{|\mathcal{O}_2(\pi)|}{|\pi|} = O(m),$$

where m is the number of variables used in π.

Since in practice in most cases $|\pi| \gg m$, the obfuscating transformation $\mathcal{O}_2$ can be accepted as *cheap* according to the Taxonomy of Obfuscating Transformations [8].

5. Conclusion

In this paper we demonstrate that one could estimate formally a resilience of obfuscating transformations against possible re-engineering activity based on static analysis algorithms (points-to analysis and static slicing). We think that our approach can be extended to the cases when more involved program analysis algorithms are applied and this will be our line of further research.

References

[1] Amoroso E.G. *Fundamentals of Computer Security Technology*. Englewood Cliffs, NJ: Prentice Hall PTR, 1994.

[2] Andersen L.O.,Program Analysis and Specialization for the C Programming Language, DIKU, University of Copenhagen, May 1994.

[3] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vedhan S., Yang K., On the (Im)possibility of obfuscating programs. *CRYPTO'01 — Advances in Cryptology*, Lecture Notes in Computer Science, **2139**, 2001, p. 1-18.

[4] Bruns G., Chandra S., Searching for points-to analysis, *IEEE Transactions on software engineering*, v. 29, 2003, N 10, p. 883-897.

[5] Chakaravarthy V.T., Horwitz S., On the non-approximatibility of points-to analysis, *Acta Informatica*, v. 38, 2002, p.587-598.

[6] Chernov A.V. On the analysis of obfuscating transformations, *Proceedings of ISP RAN*, v. 3, 2002, p. 7-39. (in Russian)

[7] Chow S., Gu Y., Johnson H., Zakharov V., An approach to the obfuscation of control flow of sequential computer programs. *Information Security Conference*, Lecture Notes in Computer Science, **2200**, 2001, p. 144-156.

[8] Collberg C., Thomborson C., Low D., A taxonomy of obfuscating transformations, Tech. Report, N 148, Dept. of Computer Science, Univ. of Auckland, 1997.

[9] Collberg C., Thomborson C., Low D., Manufacturing cheap, resilient and stealthy opaque constructs. *Symposium on Principles of Programming Languages*, 1998, p.184-196.

[10] Collberg C., Thomborson C., Low D. Breaking abstraction and unstructuring data structures. *IEEE International Conference on Computer Languages*, 1998, p.28-38.

[11] Collberg C., Thomborson C., Watermarking, tamper-proofing and obfuscation — tools for software protection. *IEEE Transactions on Software Engineering*, **28**, N 2, 2002, p. 735-746.

[12] Devanbu P.T., Stubblebine S. Software engineering for security: a roadmap. *Future of SE Track*, 2000, 227-239.

[13] Gollmann D. *Computer Security*. New York: Willey, 1999.

[14] Hada S., Zero-knowledge and code obfuscation. *ASIACRYPT'2000 — Advances in Cryptology*, 2000.

[15] Heintze N., Tardieu O., Ultra-fast alias analysis using CLA, in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[16] Horwitz S., Shapiro M., Fast and Accurate Flow-Insensitive Points-To Analysis, , CS Dept., University of Wisconsin-Madison.

[17] Horwitz S., Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, **19**, N 1, 1997, p. 1-6.

[18] Horwitz S., Reps T., Binkley D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, v.12, No 1, p. 26-60.

[19] Howard M., LeBlank D., Writing Secure Code, Microsoft Press, 2003.

[20] Landi W. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, **1**, N 4, 1992, p. 323-337.

[21] Linn C., Debray S., Obfuscation of executable code to improve resistance to static disassembly, in *Proceedings of the 10th ACM conference on Computer and communication security*, 2003, p.290–299.

[22] MacDonald J., On Program Security and Obfuscation, Technical Report, University of California, 1998.

[23] Steensgaard B., Points-to analysis in almost linear time, March 1995.

[24] Ogisso T., Sakabe Y., Sochi M., Miyaji A. Software obfuscation on a theoretical basis and its implementation, *IEEE Transactions on Fundamentals*, E86-A(1), 2003.

[25] Ottenstain K.J., Ottenstain L.M., The program dependence graph in a software development environment, *ACM SIGPLAN Notices*, v. 19, 1984, N 5, p. 177-184.

[26] Ramalingam G., The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, **16**, N 5, 1994, p. 1467-1471.

[27] Varnovsky N.P., Zakharov V.A., On the possibility of provably secure obfuscating programs, in *Proceedings of Andrei Ershov Fifth International Conference "Perspectives of System Informatics"*, Novosibirsk, 2003, p.76–86.

[28] Wang C., Hill J., Knight J. Davidson J., Software tamper resistance: obstructing static analysis of programs, Tech. Report, N 12, Dep. of Comp. Sci., Univ. of Virginia, 2000.

[29] Weiser M., Program slicing, *IEEE Transactions of Software Engineering*, v. 10, 1984, N 4, p. 352-357.

[30] Wroblewski G., General method of program code obfuscation, in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2002, p. 153-159.