Control Code Obfuscation by Abstract Interpretation

Mila Dalla Preda and Roberto Giacobazzi
Dipartimento di Informatica Università di Verona
Strada Le Grazie 15, 37134 Verona (Italy)
E-mail: dallapre@sci.univr.it — roberto.giacobazzi@univr.it

***Abstract*— Control code obfuscation is intended to prevent malicious reverse engineering of software by masking the program control flow. These obfuscating transformations often rely on the existence of opaque predicates, that support the design of transformations that break up the program control flow. We prove that an algorithm for control obfuscation by opaque predicate insertion can be systematically derived as an abstraction of a suitable semantic transformation. In this framework, deobfuscation is interpreted as an attacker which can observe the computational behaviour of programs up to a given precision degree. Both obfuscation and deobfuscation can therefore be interpreted as approximations of program semantics, where approximation is formalized using abstract interpretation theory. In particular we prove that abstract interpretation provides here the adequate setting to measure the potency of an obfuscation algorithm by comparing the degree of abstraction of the most abstract domains which are able to disclose opaque predicates.**
Keywords: Code Obfuscation, Abstract Interpretation, Program Transformation, Program analysis, Semantics.

I. Introduction

Code obfuscation is one of the most promising techniques to prevent malicious reverse engineering of software [1], [2], [3], [4]. The goal of code obfuscation is to covert a program into an equivalent one (up to some notion of observational equivalence) which is more hard to understand and reverse engineer. The obfuscating transformations can be classified according to the kind of information they target [3]. *Layout* transformations remove source code formatting and scramble identifiers. *Control* transformations affect the control flow of the program, while *data* transformations operate on the data structures used in the program [26]. The aim of control obfuscators is to obscure the code by affecting its control flow, in order to make it more difficult to understand for a malicious user (attacker). Examples are: *control aggregation transformation* changing the grouping of statements, *loop unrolling*, *control ordering transformations* altering the order of statement execution and *control computation transformers* hiding the true control flow of programs (see [3] for an excellent taxonomy). These latter syntactic transformations often rely on the existence of *opaque predicates*. A predicate is opaque if its value is known a priori to the obfuscation, but this value is difficult for the deobfuscator to deduce [4]. Given such opaque predicates, it is possible to construct transformations that break up the flow-of-control of the program by inserting dead or buggy code in branching guided by opaque predicates. In this case the resilience of the transformation clearly depends upon the resilience of the opaque predicate. Typically P^T and P^F denote the opaque predicate that evaluates respectively always to *true* and *false*. Fig. 1 illustrates an example of

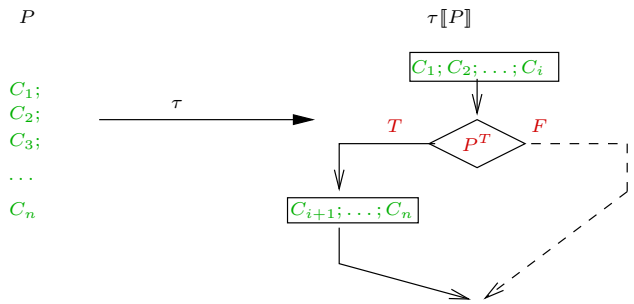


Fig. 1. Opaque predicate insertion

how the insertion of an opaque predicate P^T can be used in order to confuse the control flow of a given program P. In fact the set of possible behaviours of the transformed program is an approximation of the set of possible behaviours of the original one. By denoting $\tau[\![P]\!]$ the obfuscation of program P, then by ignoring conditional statements, $\tau[\![P]\!]$ admits more computational flows than what P does. These "spurious" additional traces represent precisely what confuses the malicious user, making the reverse engineering of $\tau[\![P]\!]$ harder. It is well known that evaluating conditional statements is a major challenge for static program analysis, and MOP approximations often rely on ignoring conditional statements in to order to make static analysis of large size code efficient [16]. Recently, the combination of static and dynamic analysis of code seems to provide a set of adequate heuristics for disclosing opaque behaviors. Experimental results are reported in [7], [33].

Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)
0-7695-2435-4/05 $20.00 © 2005 IEEE

The Problem

One of the major drawback of the existing control obfuscation techniques is that, to the best of our knowledge, there isn't a formal definition of the relation between control obfuscation algorithms and their effects on the formal semantics of programs. Because a reasonable assumption is that any intelligent attack requires a possibly partial knowledge of program semantics [5], a stronger link between program semantics and obfuscation algorithms would provide more effective transformations, where the insertion of opaque predicates can be driven by the semantics of the program, *e.g.,* by injecting misleading branching where static program analysis may fail. The lack of a formal definition of attack models, *i.e.,* of the knowledge of the the transformed program semantics by malicious users, is also a weakness of the existing techniques. In fact in this way it is hard to provide a uniform metric where obfuscated code can be compared with respect to resilience and potency. In this paper we consider the problem of both systematically deriving obfuscation algorithm for opaque predicates insertion from semantic transformations and modeling attackers for comparing control obfuscation methods.

Main Results

Abstract interpretation is a general theory introduced by Cousot & Cousot [15], [16] for describing the relationship among semantics of programming languages at different levels of abstraction. This general methodology for describing and formalizing approximate computations finds interesting applications in different areas of computer science, like for instance in model checking [8], [19], [25], verification of distributed memory systems [23], process calculi [9], [34], security [28], [29], type inference [13], [27], theorem proving [31], constraint solving [6], comparative semantics [10], [14], [17], [20]. In this paper we consider abstract interpretation [15] as a general framework for both specifying obfuscation algorithms and for modeling attackers. This main idea is based on the fact that abstract interpretation is general enough to model both static program analysis as well as any, possibly non-decidable, approximations of program semantics [11], [14]. In the design of obfuscation algorithms we exploit a recent result in [18], where the syntax of a program is considered as an abstraction of its semantics, since the semantics of a program contains more information, *i.e.,* it is more precise, than its syntax. Moreover the authors give a semantic-based formalization of syntactic program transformation has as an approximation, by abstract interpretation, of the corresponding semantic transformation. This allows us to systematically derive syntactic obfuscation algorithms by abstracting semantic transformations. In the case of control code obfuscation, obfuscating corresponds precisely to allow spurious traces in abstract semantics. Spuriousness cause a loss of precision which can be specified as an abstract interpretation of program semantics. The semantic transformation injecting opaque predicates is then defined in fix-point form, from which an iterative algorithm for syntactic code obfuscation can be systematically derived. In this work we model attackers as an approximation of the program semantics, in particular each attacker is modeled by the abstract domain that captures the property of the program semantics it wishes to grasp. In this framework we prove that the relation between semantic obfuscation and deobfuscation is a Galois connection and that the resilience of the obfuscator strictly depends upon the resilience of the inserted opaque predicates. The more concrete is the semantics needed for a precise evaluation of the opaque predicate the more high is the degree of resilience of the obfuscation. Breaking opaque predicates is therefore essential for distinguishing spurious behaviors from concrete ones. Once again, abstract interpretation plays a key role here: The degree of abstraction of the malicious user corresponds precisely to the power of the attacker. Attackers by static program analysis are specified as decidable abstractions of semantics, while attackers by dynamic program analysis correspond to possibly non-decidable abstractions. We prove that breaking opaque predicates corresponds to have complete abstractions, in the sense of abstract interpretation, as models of attackers. In the abstract interpretation framework completeness means that no loss of precision is accumulated by approximating opaque predicates in the abstract domain [22], *i.e.,* that the computation of the semantics of the program in the abstract domain leads to precise knowledge of the value of the opaque predicate. In literature, given an incomplete abstract domain, it has been defined a formal technique for systematically derive the more abstract domain that refines it making it complete [22]. Therefore, given an attacker, the most abstract domain which refines the attacker and is complete for the given opaque predicates specifies the borderline between harmless and effective deobfuscators. Completeness refinement can be used to measure resilience: A coarser refinement refers to a weaker opaque predicate. This provides a way for measuring resilience of control obfuscation by comparing the complete abstractions for the injected opaque predicates in the lattice of abstract interpretations.

II. Preliminaries

A. Semantics

The semantics of a program is a formalization of its behaviours for every possible input. Let Σ be a nonempty set of states and $\frown \subseteq \Sigma \times \Sigma$ be a transition relation, then Σ^+ and $\Sigma^\omega \stackrel{\text{def}}{=} \mathbb{N} \to \Sigma$ denote respectively the finite nonempty and the infinite sequences of states, moreover $\Sigma^\infty \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$. Given a sequence $\sigma \in \Sigma^\infty$ of length $|\sigma| \in \mathbb{N} \cup \omega$, we denote with σ_i the i-th element of such sequence. A *trace* is a sequence of states $\sigma = \sigma_0 \frown \sigma_1 \frown \ldots \frown \sigma_n$ in transition relation, i.e., for all $i < |\sigma| : \langle \sigma_i, \sigma_{i+1} \rangle \in \frown$. For rea

Arithmetic expressions	$E \in \mathbb{E}$	$E ::= n \mid X \mid E_1 - E_2$
Boolean expressions	$B \in \mathbb{B}$	$B ::= E_1 < E_2 \mid \neg B_1 \mid B_1 \vee B_2 \mid tt \mid ff$
Program actions	$A \in \mathbb{A}$	$A ::= B \mid X := E \mid X :=?$
Commands	$C \in \mathbb{C}$	$C ::= L_1 : A \to L_2$ where $L_1, L_2 \in \mathbb{L} \cup \{l\}$
Programs		$\mathbb{P} \stackrel{\text{def}}{=} \wp(\mathbb{C})$

TABLE I

ABSTRACT SYNTAX OF THE IMPERATIVE LANGUAGE

Arithmetic expressions: $\mathbf{A} : \mathcal{E}[\![\chi]\!] \to D_\Upsilon$ and $var[\![E]\!] \subseteq \chi$

$\mathbf{A}[\![n]\!]\rho \stackrel{\text{def}}{=} n \quad \mathbf{A}[\![X]\!]\rho \stackrel{\text{def}}{=} \rho(X)$

$\mathbf{A}[\![E_1 - E_2]\!]\rho \stackrel{\text{def}}{=} \mathbf{A}[\![E_1]\!]\rho - \mathbf{A}[\![E_2]\!]\rho$

Boolean expressions: $\mathbf{B} : \mathcal{E}[\![\chi]\!] \to \{\mathit{t}, \mathit{f}, \mathit{\Lambda}\}$ and $var[\![B]\!] \subseteq \chi$

$\mathbf{B}[\![E_1 < E_2]\!]\rho \stackrel{\text{def}}{=} \mathbf{A}[\![E_1]\!]\rho < \mathbf{A}[\![E_2]\!]\rho$

$\mathbf{B}[\![B_1 \vee B_2]\!]\rho \stackrel{\text{def}}{=} \mathbf{B}[\![B_1]\!]\rho \vee \mathbf{B}[\![B_2]\!]\rho$

$\mathbf{B}[\![\neg B]\!]\rho = \neg \mathbf{B}[\![B]\!] \quad \mathbf{B}[\![\mathit{\Lambda}]\!]\rho \stackrel{\text{def}}{=} \mathbf{B}[\![\dot{f}]\!]\rho \stackrel{\text{def}}{=} \dot{f}$

Program actions: $\mathbf{S} : \mathbb{A} \to (\mathcal{E}[\![\chi]\!] \to \wp(\mathcal{E}[\![\chi]\!]))$ and $var[\![A]\!] \subseteq dom(\rho)$

$\mathbf{S}[\![B]\!]\rho \stackrel{\text{def}}{=} \{\ \rho' \mid \mathbf{B}[\![B]\!]\rho' = \wedge\rho' = \rho\ \}$

$\mathbf{S}[\![X :=?]\!]\rho \stackrel{\text{def}}{=} \{\ \rho' \mid \exists z \in \mathbb{Z} : \rho' = \rho[X := z]\ \}$

$\mathbf{S}[\![X := E]\!]\rho \stackrel{\text{def}}{=} \rho[X := \mathbf{A}[\![E]\!]\rho]$

$\mathbf{S}[\![]\!]\rho \stackrel{\text{def}}{=} \mathbf{S}[\![\texttt{skip}]\!]\rho \stackrel{\text{def}}{=} \{\rho\}$

TABLE II

SEMANTICS OF THE IMPERATIVE LANGUAGE

$\frown$ will be sometimes omitted and a sequence of states will have the simplest form $\sigma_0\sigma_1 \ldots \sigma_n$. The sequence μ is a *subtrace* of σ if there exists $i, j \in [0, |\sigma|[$, where $i < j$ and $\mu = \sigma_i \frown \sigma_{i+1} \ldots \frown \sigma_j$. The *maximal trace semantics* of the transition system $\langle \Sigma, \frown \rangle$ is $\mathcal{T}^\infty \stackrel{\text{def}}{=} \mathcal{T}^+ \cup \mathcal{T}^\omega$, where $\mathcal{T}^+$ is the set of finite traces and $\mathcal{T}^\omega$ the set of infinite traces of $\langle \Sigma, \frown \rangle$ [14]. From now on the (concrete) semantics $S[\![P]\!]$ of a given program P is intended as the trace semantics $\mathcal{T}^\infty$ of P. In [14] Cousot defines a hierarchy of semantics, where approximated semantics can be derived as successive abstract interpretations of the maximal trace semantics. In the following we consider a simple imperative language defined in [18] with abstract syntax in Table I, where $L_1 : A \to L_2$ is a general command C, and a program is defined as a set of commands. The following basic functions are then defined:

$lab[\![L_1 : A \to L_2]\!] \stackrel{\text{def}}{=} L_1$	$lab[\![P]\!] \stackrel{\text{def}}{=} \{lab[\![C]\!] \mid C \in P\}$
$var[\![L_1 : A \to L_2]\!] \stackrel{\text{def}}{=} var[\![A]\!]$	$var[\![P]\!] \stackrel{\text{def}}{=} \cup_{C \in P}\, var[\![P]\!]$
$succ[\![L_1 : A \to L_2]\!] \stackrel{\text{def}}{=} L_2$	$act[\![L_1 : A \to L_2]\!] \stackrel{\text{def}}{=} A$

A program variable $X \in var[\![P]\!]$ takes its value in a given semantic domain D. An *environment* $\rho \in \mathcal{E}$ is a map from variables $X \in dom(\rho)$ to values $\rho(X) \subseteq D$, which specifies the values of program variables. The semantics of boolean and arithmetic expressions, together with the semantics of program actions of this programming language are defined in Table II. A *state* is specified as a pair $\langle \rho, C \rangle$, where C is the next command to be executed in the environment ρ. The *transition relation* between states specifies the set of states that can be reached from a given state: $\frown_P (\langle \rho, C \rangle) \stackrel{\text{def}}{=} \{(\rho', C') \mid \rho' \in \mathbf{S}[\![act(C)]\!]\rho,\ succ[\![C]\!] = lab[\![C']\!],\ \rho, \rho' \in \mathcal{E}[\![P]\!],\ C' \in P\}$. The *finite partial traces semantics* $S[\![P]\!]$ of program $P \in \mathbb{P}$ is computed as the least fix-point of the function $F[\![P]\!](X) \stackrel{\text{def}}{=} X \cup \{\sigma_i \frown \sigma_i' \frown \sigma \mid \sigma_i' \in \frown_P (\sigma_i),\ \sigma_i' \frown \sigma \in X\}$. Note that $F[\![\varnothing]\!]$ is exactly the set of terminal blocking states. In general the trace semantics of a program is a subset of Σ^*. Therefore the set $\wp(\Sigma^\infty)$ of all possible subsets of Σ^*, is the set of all possible semantics. Considering the inclusion order $\subseteq$, we have that $\langle \wp(\Sigma^\infty), \subseteq \rangle$ is a complete lattice.

B. Abstract Interpretation

According to a widely recognized definition: *"Abstract interpretation is a general theory for approximating the semantics of dynamic systems"* [12]. It was originally developed by Cousot & Cousot [15] as a unifying framework for specifying and then validating static program analysis. The basic idea of abstract interpretation is that the behaviour of a program at different levels of abstraction is an approximation of its (concrete) semantics. Therefore abstract interpretation provides a formal method to approximate semantics. The (concrete) semantics of a program is computed on the (concrete) domain, *i.e.,* the set of mathematical objects on which the program runs. Abstract domains are given by the set of properties of these objects, and are related to each other w.r.t. their relative degree of precision. In the standard Cousot & Cousot's abstract interpretation theory, abstract domains can be specified either by Galois connections, i.e. adjunctions, or by upper closures operators [15], [16]. Let $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ be a complete lattice, where the set C is equipped with the ordering relation $\leq$, and for any $S \subseteq C$: $\vee S$ is the *lub* of S, and $\wedge S$ is the *glb* of S, $\top$ is the greatest element while $\bot$ the least element. The downward closure of $S \subset C$ is $\downarrow S \stackrel{\text{def}}{=} \{x \in C \mid \exists y \in S. x \leq y\}$, and for $x \in C$, $\downarrow x$ is a shortland for $\downarrow \{x\}$. When two domains C and A are related by a pair of adjoint maps $\alpha : C \to A$ and $\gamma : A \to C$, then (C, α, A, γ) is a *Galois connection* (GC). Two functions α and γ form an adjunction when for all $a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$, in this case α is the abstraction map while γ is the concretization map, and C and A are respectively the concrete and abstract domain. An *upper closure operator* (closure) on C is an operator $\mu : C \to C$ monotone, idempotent and extensive. We denote with *uco*(C) the set of all possible closures on C. When C is a complete lattice then also $\langle$*uco*$(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.\bot \rangle$ is a complete lattice where $\mu \sqsubseteq \mu'$ if and only if $\mu'(C) \subseteq \mu(C)$, meaning that the closure μ is more concrete than the closure μ' when it contains more information. Each closure $\mu \in$ *uco*(C) is uniquely determined by the set of its fix-points $\mu(C)$. $X \subseteq C$ is a set of fix-points of closure if and only if X is a *Moore family* of C, i.e., $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$, where $\wedge\varnothing = \top \in \mathcal{M}(X)$, if and only if X is isomorphic to an abstract domain A in a GC (C, α, A, γ), i.e. $A \cong \mu$

$\iota : \mu(C) \to A$ and $\iota^{-1} : A \to \mu(C)$ being an isomorphism. In this case $(C, \iota \circ \mu, A, \iota^{-1})$ is a GC, where $\mu = \gamma \circ \alpha$. Therefore *uco*$(C)$ is isomorphic to the so called *lattice of abstract interpretations* of C [16]. In this case $A \sqsubseteq B$ if and only if $B \subseteq A$ as Moore families of C, if and only if A is more concrete than B, *i.e.,* A contains more informations than B.

Let us consider the function $f : C^n \to C$, and the abstract domain $\mu \in$ *uco*(C). The abstraction μ is *sound* for f if $\mu \circ f \sqsubseteq \mu \circ f \circ \langle \mu \dots \mu \rangle$ [16]. This means that an abstract domain is sound, *i.e.,* correct, for a concrete function, if the computation of the abstract function on the abstraction of the input is an approximation of the abstraction of the computation of the concrete function on the original input. Meaning that there is a possible loss of precision by computing the function on the abstract domain. This notion of soundness can be expressed, by adjunction, also by the following relation $f \circ \mu \sqsubseteq \mu \circ f \circ \langle \mu \dots \mu \rangle$. Given an abstract domain $\mu \in$ *uco*(C), then $f^{\mu} \stackrel{\text{def}}{=} \mu \circ f \circ \mu$ is the *best correct approximation* (*bca*) of f in μ [16]. When the soundness conditions are satisfied with equality we have two different notions of *completeness*. When $\mu \circ f = \mu \circ f \circ \langle \mu \dots \mu \rangle$, *i.e.,* when no loss of precision is accumulated in the abstract computation, the closure μ is said to be *(backward)* $\mathcal{B}$*-complete* for f. On the other side when $f \circ \mu = \mu \circ f \circ \langle \mu \dots \mu \rangle$ we have that μ is *(forward)* $\mathcal{F}$*-complete* for f, meaning that no loss of precision is accumulated by approximating the result of the function f on μ. It has been proved that $\mathcal{B}(\mathcal{F})$-completeness is a domain property, namely that it only depends upon the structure of the underlying abstract domain [22]. In [22] and [21] the authors give a systematic way for minimally transforming an abstract domain in order to make it complete for a given function. In this work we are particularly interested in $\mathcal{F}$-completeness. An abstract domain μ is $\mathcal{F}$-complete for f if and only if $\forall \bar{x} \in \mu(C^n)$ then $f(\bar{x}) \in \rho(C)$. If F is a (finite) set of functions, then

$$\mathcal{R}_F(\mu) \stackrel{\text{def}}{=} \mathit{gfp}(\lambda X.\ \mu \sqcap \mathcal{M}(\bigcup_{f \in F} f(X)))$$

is the most abstract domain extending μ and making it $\mathcal{F}$-complete for f [22], [21].

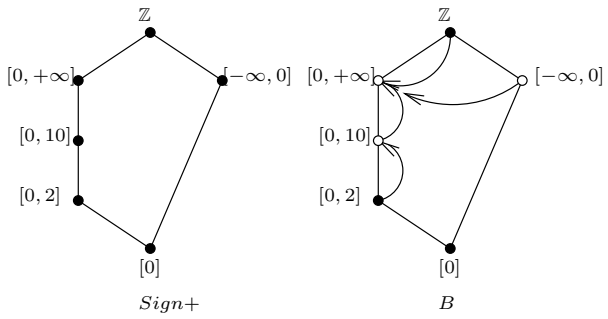


Fig. 2. The abstract domain $Sign+$ and its abstraction B

Example 1: The domain $Sign+$ in Fig. 2 is an abstraction of $\langle \wp(\mathbb{Z}), \subseteq \rangle$, let $sq : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ be the square operation defined as follows: $sq(X) = \{x^2 \mid x \in X\}$, for $X \in \wp(\mathbb{Z})$. Let $\mu \in$ *uco*$(\wp(\mathbb{Z}))$ be the closure operator associated with $Sign+$, where the abstraction and concretization maps are the most obvious ones. Let $sq^\sharp : Sign+ \to Sign+$ be the *bca* of sq in $Sign+$, such that $sq^\sharp(X) = \mu(sq(X))$ where $X \in Sign+$. Let μ_b be the closure operator associated with the abstract domain B, which is an abstraction of $Sign+$. The arrows in the abstract domain B represent the function $sq^\sharp$ when we are not in a fix-point. It easy to see that the upper closure operator μ_b is not $\mathcal{F}$-complete for $sq^\sharp$. In fact $\mu_b(sq^\sharp(\mu_b([0,2]))) = \mathbb{Z}$, while $sq^\sharp(\mu_b([0,2])) = [0,10]$. It is possible to observe that μ_b is not $\mathcal{F}$-complete because it does not contain the image of $sq^\sharp$, *i.e.,* the points $[0,10]$ and $[0,+\infty]$. On the other side considering the abstract domain A

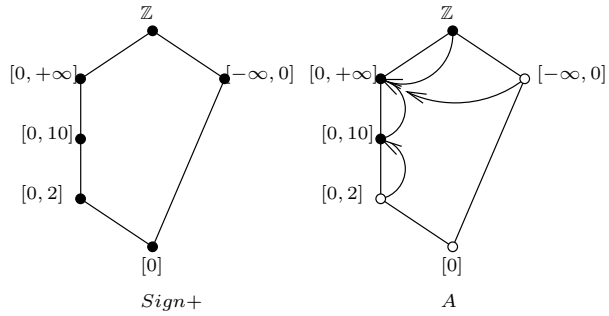


Fig. 3. The abstract domain $Sign+$ and its abstraction B

in Fig. 3, let $\mu_a = \{\mathbb{Z}, [0,+\infty], [0,10]\}$ be the corresponding upper closure operator. In this case we have that μ_a is not $\mathcal{B}$-complete for $sq^\sharp$, in fact $\mu_a(sq^\sharp(\mu_a([0])))) = [0,+\infty]$ while $\mu_a(sq^\sharp([0])) = [0,10]$. We can observe that μ_a is not $\mathcal{B}$-complete because it does not include the maximal inverse image of $sq^\sharp$, namely the point $[0,2]$. In fact by adding the maximal inverse images of a given function to the abstract domain, we make it $\mathcal{B}$-complete for that function [22].

C. Program Transformation

An interesting application of abstract interpretation is the one proposed by Cousot & Cousot in [18], where abstract interpretation is investigated in the field of program transformation. In general a program transformation is a meaning-preserving mapping on programming language [30]. Following this observation in [18] the authors start to introduce a general framework for reasoning on semantics-based program transformations. In particular they develop a constructive design methodology for systematically derive syntactic transformations by approximation of a semantic transformation. In this work the authors consider the syntax of programs as an abstraction of their semantics, and they formalize this idea by the GC $(\langle \wp(\Sigma^\infty), \sqsubseteq \rangle, p, \langle \mathbb{P}, \leq \rangle, S)$, where $S : \mathbb{P} \to \wp(\Sigma^\infty)$ is the semantics function and $p : \wp(\Sigma^\infty) \to \mathbb{P}$ is the

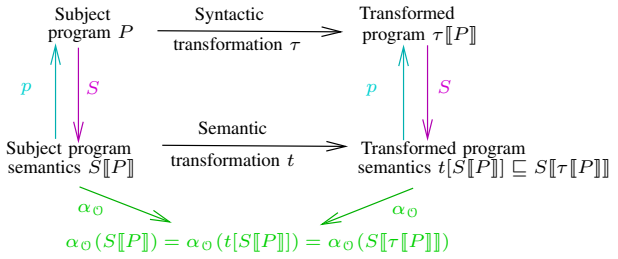


Fig. 4. Syntactic-Semantic Program Transformations

program whose semantics upper approximates $X \in \wp(\Sigma^\infty)$. Formally:

$$p[X] \stackrel{\text{def}}{=} \{C | \exists \sigma \in X; \exists i \in [0, |\sigma|[: \exists \rho \in \mathcal{E} : \sigma_i = \langle \rho, C \rangle\} \quad (1)$$

Given a program $P \in \mathbb{P}$ and a syntactic transformation $\tau : \mathbb{P} \to \mathbb{P}$, it is possible to consider the corresponding semantic transformation $t : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ that transforms the semantics of a program $S[\![P]\!]$ into the semantics of the transformed program $S[\![\tau[\![P]\!]]\!]$. A program transformation is said to be correct if at some level of abstraction it is meaning preserving. This means that a syntactic transformation τ is *correct* w.r.t. an observational abstraction $\alpha_{\mathcal{O}} \in \wp(\Sigma^\infty)$, if the transformed program is equivalent to the original program under $\alpha_{\mathcal{O}}$, i.e., for all $P \in \mathbb{P}$: $\alpha_{\mathcal{O}}[\![P]\!] = \alpha_{\mathcal{O}}[\![\tau[\![P]\!]]\!]$. The work done in [18] is summarized in Fig 4.

III. Semantic-based Control Obfuscators

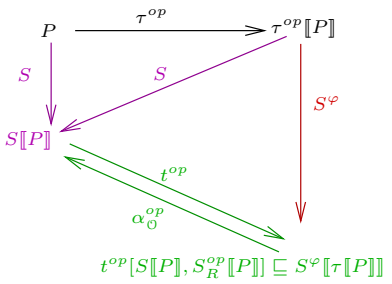
The classical definition of code obfuscation is the one of Collberg *et al.* [1], [3] where an obfuscator is defined as a transformation that given a program P returns a program P', which is observational equivalent to P but more complex, and therefore more difficult for an attacker to understand. In particular a program transformation $\tau : \mathbb{P} \to \mathbb{P}$ is said to be *potent* if given a program P in input then $\tau[\![P]\!]$ is more complex than P. Therefore an obfuscator is a potent and meaning preserving program transformation. It is clear that this definition of code obfuscation relays on the notion of potency, *i.e.,* on a metric for measuring program complexity. In order to formalize the obfuscating transformations in the above semantic field, we first need to give a semantic definition for measuring program complexity, and therefore for defining potency [32]. The idea is that a program P', obtained by a transformation of P, is more complex than a program P, when there exists at least a semantic property that is evaluated differently on P than on P'.

Definition 1: Given two programs P and P' in $\mathbb{P}$, such that P and P' are semantically equivalent, then P' is more complex then P, if there is a property $\alpha \in uco(\wp(\Sigma^\infty))$ such that: $\alpha(S[\![P]\!]) \neq \alpha(S[\![P']\!])$.

Observe that, in the previous definition, the abstract property α is computed on the concrete trace semantics of program P. Given a syntactic transformation τ, we define $\delta_\tau \stackrel{\text{def}}{=} \sqcap \{ \varphi \in uco(\wp(\Sigma^\infty)) \mid \forall P \in \mathbb{P}.\ \varphi(S[\![P]\!]) = \varphi(S[\![\tau[\![P]\!]]\!]) \}$ as the most concrete property preserved by τ. In [32] it has been proved the existence and some basic features of this property. In this field every syntactic transformation can be seen as an obfuscator if it is potent. In fact we say that $\tau : \mathbb{P} \to \mathbb{P}$ is a δ-obfuscator if $\delta = \delta_\tau$ and the transformation τ is potent. Once again this means that every potent program transformation acts as an obfuscator. For instance the well known constant propagation transformation for code optimization can be seen as an obfuscation that hides some of the properties of actions, such as distinguishing whether an action is an assignment or a null action [32]. This because, by constant propagation, assignments to constant variables are removed. The following result shows how the potency of a syntactic transformation can be deduced by analyzing the corresponding semantic transformation.

Lemma 1: Given a semantic transformation $t : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ if there exists a property α and a program P such that: $\alpha(S[\![P]\!]) \neq \alpha(t(S[\![P]\!]))$ then the corresponding syntactic transformation $\tau : \mathbb{P} \to \mathbb{P}$ is potent.

In the following we define both the semantic transformation t^{op} performing opaque predicate insertion, and the observational abstraction. Then by using the results of [18] we derive the corresponding syntactic transformation τ^{op}. We denote with $\alpha^{op}_{\mathcal{O}}$ the abstract semantics preserved by the transformation. In this particular case it turns out that the trace semantics of the original and obfuscated program are the same. This happens because when the semantics executes the test on the opaque predicate P^T this returns always *true*, and therefore the *false* branch is never taken. This means that when the attacker has access to the trace semantics of the obfuscated program, it actually knows the trace semantics of the original program. Something different happens if we consider abstract domains as models of the malicious users. In this case the attacker $\varphi \in uco(\wp(\Sigma^\infty))$ computes the semantics of the obfuscated program on its abstract domain φ and only if this domain is complete for the opaque predicate it will be able to disclose the obfuscation. In the following we are going to investigate which are the attackers defeated by the insertion of a particular opaque predicate. The following figure represents an overview of what we are going to prove.



A. Semantic Transformation

We first define the preliminary static analysis $S^{op} : \mathbb{P} \to \wp(\mathbb{C})$ that, given a program, returns the set of program points where it is possible to insert an opaque predicate. More precisely the preliminary static analysis returns the subset $S^{op}[\![P]\!]$ of commands of P, which has cardinality equal to the number of opaque predicates we want to add to the subject program. These are the program points where static analysis of opaque predicates may fail. The semantic transformation that provides the opaque predicate insertion according to S^{op}, is specified by the following definition.

Definition 2: Given a program $P \in \mathbb{P}$ and a preliminary static analysis $S^{op}[\![P]\!]$, $t^{op} : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ is defined as:

$$\begin{array}{rcl} t^{op}[S[\![P]\!], S^{op}[\![P]\!]] & \stackrel{def}{=} & \{\ t^{op}[\sigma, S^{op}[\![P]\!]] \mid \sigma \in S[\![P]\!]\ \} \\ t^{op}[\sigma, S^{op}[\![P]\!]] & \stackrel{def}{=} & \lambda i.t^{op}[\sigma_i, S^{op}[\![P]\!]] \end{array}$$

where for each state σ_i we have two possible cases: If $C_i \notin S^{op}[\![P]\!]$ then $t^{op}[\sigma_i, S^{op}[\![P]\!]] = \sigma_i$; otherwise $t^{op}[\sigma_i, S^{op}[\![P]\!]] = \{\langle \rho_i, L_i : A_i \to \tilde{L}_i\rangle, \langle \rho_i, \tilde{L}_i : P^T = tt \to L_i\rangle, \langle \rho_i, \tilde{L}_i : P^T = ff \to L_S\rangle\}$.

We consider $\eta^\frown\langle \rho_i, L_i : A_i \to L'_i\rangle$ as the trace obtained by folding $\eta^\frown\langle \rho_i, L_i : A_i \to \tilde{L}_i\rangle^\frown\langle \rho_i, \tilde{L}_i : P^T = tt \to L'_i\rangle$.

Example 2: Given a program P let us consider the trace $\sigma = \langle \rho_0, C_0\rangle^\frown\langle \rho_1, C_1\rangle^\frown\langle \rho_2, C_2\rangle^\frown\langle \rho_3, C_3\rangle^\frown\langle \rho_4, C_4\rangle$, where $\sigma \in S[\![P]\!]$. Recall that the command of a general state σ_i is of the form $C_i = L_i : A_i \to L'_i$ and let us assume that $S^{op}[\![P]\!] = \{C_1, C_3\}$. By computing the semantic transformation on σ we obtain that $t^{op}[\sigma, S^{op}[\![P]\!]]$ is given by the following set of three traces (see Fig. 5):

$$\left\{\begin{array}{l} - \ \langle \rho_0, C_0\rangle\langle \rho_1, L_1 : A_1 \to \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : P^T = tt \to L'_1\rangle\langle \rho_2, C_2\rangle \\ \quad \langle \rho_3, L_3 : A_3 \to \tilde{L}_3\rangle\langle \rho_3, \tilde{L}_3 : P^T = tt \to L'_3\rangle\langle \rho_4, C_4\rangle) \\ - \ \langle \rho_0, C_0\rangle\langle \rho_1, L_1 : A_1 \to \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : P^T = ff \to L_S\rangle \\ - \ \langle \rho_0, C_0\rangle\langle \rho_1, L_1 : A_1 \to \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : P^T = tt \to L'_1\rangle\langle \rho_2, C_2\rangle \\ \quad \langle \rho_3, L_3 : A_3 \to \tilde{L}_3\rangle\langle \rho_3, \tilde{L}_3 : P^T = ff \to L_S\rangle \end{array}\right.$$

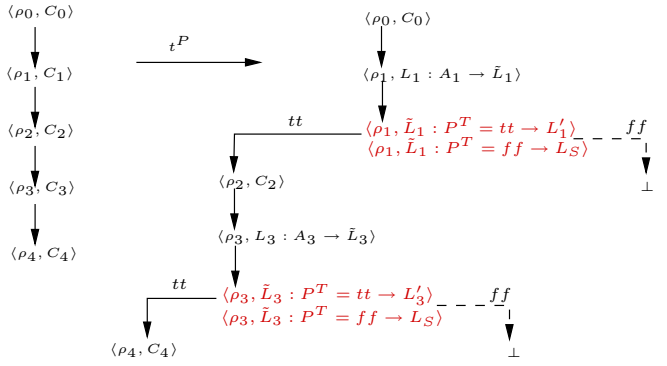


Fig. 5. An example of semantic obfuscation

This example shows how the insertion of opaque predicates confuses the flow of control of the original program. The semantic transformation t^{op} looses precision with respect to the input semantics, because it adds "false" traces (noise) to the original semantics. The following result shows that, up to trace folding, this semantic approximation is a semantic domain abstraction, namely t^{op} is a closure on $\wp(\Sigma^\infty)$.

Theorem 1: $t^{op} \in uco(\wp(\Sigma^\infty))$ up to trace folding.

B. Observational Semantics

The observational semantics that does not distinguish between the original and transformed program is given by the following definition.

Definition 3: $\alpha^{op}_{\mathcal{O}} : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ is defined as:

$$\begin{array}{rcl} \alpha^{op}_{\mathcal{O}}(X) & \stackrel{def}{=} & \{\ \alpha^{op}_{\mathcal{O}}(\sigma) \mid \sigma \in X\ \} \\ \alpha^{op}_{\mathcal{O}}(\sigma) & \stackrel{def}{=} & \epsilon^\frown\alpha^{op}_{\mathcal{O}}(\sigma) \end{array}$$

- if $act[\![C']\!] \neq P^T$ then
$\delta^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho, C\rangle^\frown\langle \rho', C'\rangle^\frown\eta) \stackrel{def}{=} \delta^\frown\langle \rho, C\rangle^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho', C'\rangle^\frown\eta)$
- if $(act[\![C']\!] = P^T \wedge P^T = tt)$ then
$\delta^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho, C\rangle^\frown\langle \rho', C'\rangle^\frown\eta) \stackrel{def}{=}$
$\delta^\frown\langle \rho, lab[\![C]\!] : act[\![C]\!] \to succ[\![C']\!]\rangle^\frown\alpha^{op}_{\mathcal{O}}(\eta)$
- if $(act[\![C']\!] = P^T \wedge P^T = ff)$ then
$\delta^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho, C\rangle^\frown\langle \rho', C'\rangle^\frown\eta) \stackrel{def}{=} \epsilon$

where ϵ is the empty trace, δ is the part of the trace already considered by the observational semantics, and η is the part of σ that has still to be observed by $\alpha^{op}_{\mathcal{O}}$.

Example 3: Let σ and $t^{op}[\sigma, S^{op}[\![P]\!]]$ be the one of Example 2. We prove that $\alpha^{op}_{\mathcal{O}}(\sigma) = \alpha^{op}_{\mathcal{O}}(t^{op}[\sigma, S^{op}[\![P]\!]])$. By applying the observational semantics to $t^{op}[\sigma, S^{op}[\![P]\!]]$, we have only one trace that differs from the empty trace ϵ: The one where $P^T = tt$ each time it occurs. In fact when in a trace there is a point in which $P^T = ff$, then the observational semantics returns the empty trace ϵ. Therefore by evaluating $\alpha^{op}_{\mathcal{O}}(t^{op}[\sigma, S^{op}[\![P]\!]])$ we obtain:

$$\left[\begin{array}{l} \langle \rho_0, C_0\rangle^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho_1, L_1 : A_1 \to \tilde{L}_1\rangle^\frown\langle \rho_1, \tilde{L}_1 : P^T = tt \to L'_1\rangle \\ \quad ^\frown\langle \rho_2, C_2\rangle^\frown\langle \rho_3, L_3 : A_3 \to \tilde{L}_3\rangle^\frown\langle \rho_3, \tilde{L}_3 : P^T = tt \to L'_3\rangle \\ \qquad\qquad ^\frown\langle \rho_4, C_4\rangle) \end{array}\right.$$
$$\left[\begin{array}{l} \langle \rho_0, C_0\rangle^\frown\langle \rho_1, L_1 : A_1 \to L'_1\rangle^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho_2, C_2\rangle^\frown \\ \langle \rho_3, L_3 : A_3 \to \tilde{L}_3\rangle^\frown\langle \rho_3, \tilde{L}_3 : P^T = tt \to L'_3\rangle^\frown\langle \rho_4, C_4\rangle) \end{array}\right.$$
$$\left[\begin{array}{l} \langle \rho_0, C_0\rangle^\frown\langle \rho_1, C_1\rangle^\frown\langle \rho_2, C_2\rangle^\frown\alpha^{op}_{\mathcal{O}}(\langle \rho_3, L_3 : A_3 \to \tilde{L}_3\rangle^\frown \\ \qquad \langle \rho_3, \tilde{L}_3 : P^T = tt \to L'_3\rangle^\frown\langle \rho_4, C_4\rangle) \end{array}\right.$$
$$\left[\begin{array}{l} \langle \rho_0, C_0\rangle^\frown\langle \rho_1, C_1\rangle^\frown\langle \rho_2, C_2\rangle^\frown\langle \rho_3, L_3 : A_3 \to L'_3\rangle^\frown \\ \qquad\qquad \alpha^{op}_{\mathcal{O}}(\langle \rho_4, C_4\rangle \\ \langle \rho_0, C_0\rangle^\frown\langle \rho_1, C_1\rangle^\frown\langle \rho_2, C_2\rangle^\frown\langle \rho_3, C_3\rangle^\frown\langle \rho_4, C_4\rangle = \sigma \end{array}\right.$$

On the other side $\alpha^{op}_{\mathcal{O}}(\sigma) = \sigma$, and therefore $\alpha^{op}_{\mathcal{O}}$ is preserved by the transformation t^{op}.

The following result proves that the observational abstraction $\alpha^{op}_{\mathcal{O}}$ is an abstraction in the sense of abstract interpretation [16] and that it is the most concrete property preserved by t^{op}.

Theorem 2: The observational abstraction $\alpha^{op}_{\mathcal{O}}$ is additive, namely $\alpha^{op}_{\mathcal{O}}(\bigvee X) = \bigvee_{Y \in X} \alpha^{op}_{\mathcal{O}}(Y)$. Moreover $\alpha^{op}_{\mathcal{O}}$ is the most concrete property preserved by the transformation t^{op}.

When the observational semantics $\alpha^{op}_{\mathcal{O}}$ is applied to the transformed program semantics, it returns exactly the original program semantics, namely $\alpha^{op}_{\mathcal{O}}(S[\![P]\!]) = S[\![P]\!]$ and therefore $\alpha^{op}_{\mathcal{O}}(t^{op}[S[\![P]\!], S^{op}[\![P]\!]]) = S[\![P]\!]$. This allows us

Opaque$(P, S^{op}[\![P]\!], P^T)$
$Q := \{C \mid C \in (P \smallsetminus S^{op}[\![P]\!])\} \cup \{\ L_S : stop \to \not{l}\ \}$
while $S^{op}[\![P]\!] \neq \emptyset$ do
$\quad Q_{new} := Q \cup \left\{ \begin{array}{l} L : A \to \tilde{L} \\ \tilde{L} : P^T = tt \to L' \\ \tilde{L} : P^T = ff \to L_S \end{array} \;\middle|\; \bar{C} \in Q : \bar{C} = \bar{L} : \bar{A} \to L, L : A \to L' \in S^{op}[\![P]\!] \right\}$
$\quad S^{op}[\![P]\!] := S^{op}[\![P]\!] \smallsetminus (Q_{new} \smallsetminus Q)$
$\quad Q := Q_{new}$
endwhile
return Q

TABLE III

THE DERIVED OBFUSCATION ALGORITHM

about observational semantics as a deobfuscator. This happens because t^{op} does not loose any of the e original program behaviours (it just adds spurious behaviors to generate confusion). Therefore, by knowing the precise evaluation of the opaque predicate P^T it is possible for $\alpha^{op}_{\mathbb{O}}$ to isolate spurious traces from the obfuscated semantics. This means that the resilience of the opaque predicate insertion transformation is proportional to the resilience of the opaque predicate itself. Therefore $\alpha^{op}_{\mathbb{O}}$ can be seen as the appropriate deobfuscator for t^{op}. Moreover $\alpha^{op}_{\mathbb{O}}$ is the inverse of t^{op} w.r.t. a program $P \in \mathbb{P}$.

Theorem 3: Given a general program $P \in \mathbb{P}$ then $\alpha^{op}_{\mathbb{O}}$ is the inverse of t^{op} w.r.t. P, namely $t^{op^{-1}} = \alpha^{op}_{\mathbb{O}}$.

C. Syntactic Transformation

Following the methodology suggested in [18]: Given a program $P \in \mathbb{P}$, the opaque semantics $t^{op}[S[\![P]\!], S^{op}[\![P]\!]]$ can be expressed in fix-point form as $lfp^{\subseteq} \lambda T.Init \cup Next(T)$, where T ranges over sets of traces, and $\sigma_i = \langle \rho_i, C_i \rangle$ is a generic program state.

$$Init \stackrel{\text{def}}{=} \left\{ \sigma_1 ^\frown \ldots ^\frown \sigma_n \;\middle|\; \begin{array}{l} n > 0 \wedge \forall i \in [1, n[\ C_i \notin S^{op}[\![P]\!] \wedge \\ C_n \in S^{op}[\![P]\!] \end{array} \right\}$$

$$Next(T) \stackrel{\text{def}}{=} \left\{ \delta\sigma_1\sigma_2 \ldots \sigma_n \;\middle|\; \begin{array}{l} \delta\sigma_1\sigma_2 \in T \wedge C_1 \in S^{op}[\![P]\!] \wedge \\ \sigma_1 = \langle \rho_1, L_1 : A_1 \to \tilde{L}_1 \rangle \wedge \\ \sigma_2 \in \{\langle \rho_1, \tilde{L}_1 : P^T = tt \to L'_1 \rangle \\ \langle \rho_1, \tilde{L} : P^T = ff \to L_S \rangle\} \wedge \\ \forall i \in [3, n[\ C_i \notin S^{op}[\![P]\!] \wedge \\ C_n \in S^{op}[\![P]\!] \end{array} \right\}$$

The set Int collects the initial subtraces of the traces of the original semantics $S[\![P]\!]$, where no state contains a command that belongs to $S^{op}[\![P]\!]$ except for the last one, meaning that these subtraces will not be affected by the obfuscating transformation. On the other side $Next$ collects the traces that begins with a general trace δ such that $\delta ^\frown \sigma_1$ where $\sigma_1 \in S^{op}[\![P]\!]$, then followed by the opaque predicate insertion, and by a subtrace $\sigma_3 ^\frown \ldots ^\frown \sigma_n$, where all the state commands do not belong to $S^{op}[\![P]\!]$ except for the last one. Iterating this computation until a fix-point is reached we obtain exactly the same set of traces given by the semantic transformation t^{op}.

Theorem 4: $lfp^{\subseteq} \lambda T.Init \cup Next(T) = t^{op}[S[\![P]\!], S^{op}[\![P]\!]]$.

This fix-point formulation of the semantic transformation $t^{op}[S[\![P]\!], S^{op}[\![P]\!]]$ naturally leads to the following iterative algorithm for control code obfuscation by opaque predicate insertion. The algorithm **Opaque** in Table III, given an input program P, the result of the preliminary static analysis $S^{op}[\![P]\!]$, and the opaque predicate P^T, returns the set of commands representing the transformed program $\tau^{op}[\![P]\!]$. The following result proves the correctness of the algorithm **Opaque**, where p is defined as in (1).

Theorem 5: **Opaque**$(P, S^{op}[\![P]\!], P^T) = p(lfp^{\subseteq} \lambda T.Init \cup Next(T))$.

Considering the control obfuscation algorithm **Opaque**, it is important to observe that the semantic of the original program P is exactly the same of the semantics of the transformed program $\tau^{op}[\![P]\!]$, namely $S[\![P]\!] = S[\![\tau^{op}[\![P]\!]]\!]$. This because the syntactic transformation inserts in the program the opaque predicate P^T, but when $S[\![\tau^{op}[\![P]\!]]\!]$ is computed, such predicate always evaluates to true, and therefore the false branch is never considered.

Corollary 1: Given a program $P \in \mathbb{P}$ and the syntactic transformation τ^{op} inserting an opaque predicate P^T, then $S[\![P]\!] = S[\![\tau^{op}[\![P]\!]]\!]$.

IV. MODELING ATTACKERS

Attackers are malicious observers of the program behaviour, whose task is to reveal semantic properties of the obfuscated program at different degrees of abstraction. This means that we can classify attackers by considering the semantic property $\varphi \in uco(\wp(\Sigma^\infty))$ they are interested in. In this setting the complete lattice $\langle uco(\wp(\Sigma^\infty)), \sqsubseteq \rangle$ is a powerful mean for comparing and classifying attackers. In particular we can say that each attacker φ that is not able to precisely evaluate the opaque predicate P^T is obfuscated by the transformation t^{op}, in fact in this case the malicious observer does not understand that the program execution always follows the true branch, and therefore it approximates the semantics of the transformed program by considering both branches. We denote with $S^{\varphi}[\![P^T]\!]$ the abstract computation of the opaque predicate, namely the calculus of the semantics of the opaque predicate on the abstract domain φ, where it may

evaluated precisely. Let $P^T : \mathbb{Z}^n \rightarrow \mathbb{B}$, where $\langle \mathbb{B}, \preceq \rangle$ is the complete lattice of boolean values. Consider the general opaque predicate $P^T(\bar{x})$ of the form $h(\bar{x}) = g(\bar{x})$, where $\bar{x} \in \mathbb{Z}^n$ and $h, g : \mathbb{Z}^n \rightarrow \mathbb{Z}$. We give the following point to point definition of equality, when the opaque predicate is defined on subsets of $\mathbb{Z}^n$.

Definition 4: Let $X \in \mathbb{Z}^n$ then $h(X) \doteq g(X)$ if and only if for all $x \in X$: $g(x) = h(x)$.

In this case $S^{\varphi}[\![P^T]\!] \stackrel{\text{def}}{=} \varphi \circ h \circ \varphi(\bar{x}) \doteq \varphi \circ g \circ \varphi(\bar{x})$, namely $S^{\varphi}[\![P^T]\!]$ is the best correct approximation of S in φ.

Theorem 6: For each attacker $\varphi \in uco(\wp(\Sigma^{\infty}))$ we have that $S^{\varphi}[\![P^T]\!] = tt$ if and only if for each program $P \in \mathbb{P}$: $\varphi(S[\![P]\!]) = \varphi(t^{op}[S[\![P]\!], S^{op}[\![P]\!]])$.

This means that the transformation t^{op} obfuscates all the attackers φ such that $S^{\varphi}[\![P^T]\!] = \top$, in fact for these attackers we have that $\varphi(S[\![P]\!]) \sqsubset \varphi(t^{op}[S[\![P]\!], S^{op}[\![P]\!]])$.

All the attackers φ that are not able to precisely evaluate the opaque predicate are defeated by the obfuscator τ^{op}, because it is not possible to know φ precisely by analyzing the transformed version of the program. It would be interesting to characterize which are the attackers defeated by a control obfuscator inserting an opaque predicate (some of the most common opaque predicate are listed in Table IV).

$\forall x, y \in \mathbb{Z}$	:	$7y^2 - 1 \neq x^2$
$\forall x \in \mathbb{Z}$	:	$3 \mid (x^3 - x)$
$\forall x \in \mathbb{N}$	:	$14 \mid (3 * 7^{4x+1} + 5 * 4^{2x-1} - 5)$

TABLE IV
OPAQUE PREDICATES

A. Attackers and Completeness.

Let us assume that each variable of the program, namely each variable in $var[\![P]\!]$, has value in the domain $\mathbb{Z}$. Let $|var[\![P]\!]| = m$, then $\langle \wp(\mathbb{Z}^m), \subseteq \rangle$ is a complete lattice. Each abstraction $\varphi \in uco(\wp(\mathbb{Z}^m))$ induces an abstraction on the values of variables and therefore on the value that the opaque predicate inputs can assume. We denote with φ also the closure in $uco(\wp(\mathbb{Z}^n))$ relative to the opaque predicate P^T, where n represents the variables involved in the opaque predicate. It may happen that φ is not precise for the evaluation of P^T, namely that $S^{\varphi}[\![P^T]\!] = \top$, while $S[\![P^T]\!] = tt$. This means that $S[\![P^T]\!] \preceq S^{\varphi}[\![P^T]\!]$. From the abstract interpretation theory [15], [16], this means that the abstract domain φ is correct but not complete for the opaque predicate P^T [16], [22]. In order for the abstract domain to be complete it must be $S[\![P^T]\!] = S^{\varphi}[\![P^T]\!]$, meaning that the evaluation of the opaque predicate on the abstract domain φ is precise. Observe that $\varphi \in uco(\wp(\mathbb{Z}^m))$ induces an abstraction on the trace semantics of programs, where each state $\langle \rho, C \rangle$ is approximated with $\langle \varphi(\rho), C \rangle$, since the environment ρ represents the values of program variables at a certain point of the computation. From now on, with abuse of notation, we will denote with φ also the abstraction induced on $\wp(\Sigma^{\infty})$. This means that φ is a general attacker for the obfuscated program.

Let us consider the general opaque predicate with the following structure $P^T(\bar{x}) : h(\bar{x}) = g(\bar{x})$, where $h, g : \mathbb{Z}^n \rightarrow \mathbb{Z}$ and $\bar{x} \in \mathbb{Z}^n$. Since the opaque predicate P^T always evaluates to true this means that for all $\bar{x} \in \mathbb{Z}^n$: $h(\bar{x}) = g(\bar{x})$. In this case we say that φ is precise for P^T if for all $\bar{x} \in \mathbb{Z}^n$ holds that $\varphi \circ h \circ \varphi(\bar{x}) \doteq \varphi \circ g \circ \varphi(\bar{x})$. We want to determine which are the conditions that guarantee that when the opaque predicate is computed on the abstract domain it is still always true. The following results shows that the precise evaluation of the opaque predicate in the abstract domain only depends upon the structure of the abstract domain (since it is related with the notion of $\mathcal{F}$-completeness of the abstract interpretation theory).

Theorem 7: Given an opaque predicate $P^T(\bar{x}) : h(\bar{x}) = g(\bar{x})$ with $\bar{x} \in \mathbb{Z}^n$, and $h, g : \mathbb{Z}^n \rightarrow \mathbb{Z}$ and $\varphi \in uco(\wp(\mathbb{Z}^n))$, then if φ is $\mathcal{F}$-complete for h and g we have that for all $\bar{x} \in \mathbb{Z}^n$: $h(\bar{x}) = g(\bar{x}) \Leftrightarrow \varphi \circ h \circ \varphi(\bar{x}) \doteq \varphi \circ g \circ \varphi(\bar{x})$.

The previous result shows that each attacker φ that is complete for the functions that define the opaque predicate, is able to defeat the obfuscators, being aware of the fact that the opaque predicate always evaluates to true. Therefore a technique to deobfuscate the transformed program in order to perform reverse engineering consists in modifying the attacker (viz., abstraction) in order to make it $\mathcal{F}$-complete for the functions composing the opaque predicate. As introduced in Section II, it is well known that there is a systematic way for minimally refining an abstract domain in order to make it $\mathcal{F}$-complete for a given function [22]. We denote by $\mathcal{R}_{P^T} : uco(\wp(\mathbb{Z}^n)) \rightarrow uco(\wp(\mathbb{Z}^n))$ the $\mathcal{F}$-completeness refinement for both the functions g and h, i.e., the abstract domain transformer $\mathcal{R}_{\{h,g\}}$. Therefore, by Theorem 7, $S^{\mathcal{R}_{P^T}(\varphi)}[\![\tau^{op}[\![P]\!]]\!] = S[\![P]\!]$, and the obfuscator τ^{op} doesn't defeat the attacker $\mathcal{R}_{P^T}(\varphi)$. This means that the *abstract domain transformation given by the completeness refinement for the opaque predicate acts as a deobfuscator*. Therefore in order to deobfuscate the transformed code it is necessary to deeply know the structure of the opaque predicate that has been used during the obfuscation, meaning that the resilience of this obfuscating techniques strictly depends on the resilience of the opaque predicate.

This approach based on modeling attackers as abstractions allows us to compare both the potency of different obfuscators and the efficiency of different attackers. For example, given an obfuscator τ^{op} that inserts the opaque predicate $P^T(\bar{x}) : g(\bar{x}) = h(\bar{x})$, let φ and ψ be two possible attackers modeled as abstract domains, that are not complete for either g or h. Assume that $\varphi \sqsubseteq \psi$ and that $\mathcal{R}_{P^T}(\psi) = \mathcal{R}_{P^T}(\varphi)$. Then τ^{op} obstructs ψ more than what φ does (Fig. 6(a)). This because the amount of information which is required in order to approximate the predicate in a complete way is

ψ than in φ. From a practical point of view, this means that the abstract interpretation (viz., static program analysis) based on the ψ abstraction needs extra help (e.g., dynamic analysis) to disclose P^T than what φ may require. On the other side,

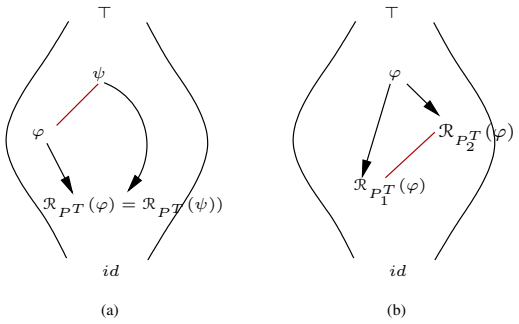


Fig. 6. (a): $\mathcal{R}_{P^T}(\psi) \sqsubset \mathcal{R}_{P^T}(\varphi)$, (b): $\mathcal{R}_{P_1^T}(\varphi) \sqsubset \mathcal{R}_{P_2^T}(\varphi)$

the same reasoning can be applied to compare the resilience of opaque predicates in the lattice of abstract interpretations. Given two opaque predicates P_1^T and P_2^T, let τ_1^{op} and τ_2^{op} be the corresponding obfuscating transformations, as derived in Section III. Given an attacker $\varphi \in \mathit{uco}(\wp(\Sigma^\infty))$ that is not complete for both g_1 or h_1 and for g_2 or h_2, then if $\mathcal{R}_{P_1^T}(\varphi) \sqsubset \mathcal{R}_{P_2^T}(\varphi)$ this means that the obfuscator τ_1^{op} is more efficient in obstructing the attacker φ than what τ_2^{op} does (Fig. 6(b)). This because the amount of information needed to make φ complete for g_1 and h_1 is larger than the one needed to make it complete for g_2 and h_2, meaning that the insertion of the opaque predicate P_1^T obstructs φ more than the insertion of P_2^T. In order to understand which obfuscation (i.e., opaque predicate) is more efficient for obstructing a given semantic property, it is necessary to compute the fix-point solution of the completeness refinement with respect to the corresponding opaque predicate of the attacker domain. The closer is the refined attacker to the identical abstraction (concrete semantics), the higher is the resilience of the obfuscation. In the limit case, if $\mathcal{R}_{P^T}(\varphi) = \mathit{id}$ then no approximation of the concrete semantics may guarantee a correct disclosure of P^T.

B. An Example

Let us consider a classical always true opaque predicate $7y^2 - 1 \neq x^2$, and the abstract domain, *i.e.,* the attacker, given by the domain of congruences on $\mathbb{Z}$, defined by the closure $C_\mathbb{Z} \in \mathit{uco}(\wp(\mathbb{Z}))$ [24]. In particular $C_\mathbb{Z}(\wp(\mathbb{Z})) = C(\mathbb{Z})$ where $C(\mathbb{Z}) = \{\varnothing\} \cup \bigcup_{n\in\mathbb{N}} \mathbb{Z}/n\mathbb{Z}$, and

$$\mathbb{Z}/n\mathbb{Z} = \begin{cases} \{\ \{x\} \mid\ x \in \mathbb{Z}\ \} & \text{if } n = 0 \\ \{\ a + n\mathbb{Z} \mid\ a = 0, ..., n-1\ \} & \text{otherwise} \end{cases}$$

which is proved to be a Moore family of $\wp(\mathbb{Z})$. Note that $\mathbb{Z}/1\mathbb{Z} = \mathbb{Z}$, $\mathbb{Z}/2\mathbb{Z} = \{2\mathbb{Z}, 1+2\mathbb{Z}\}$, etc. In this case we observe that $C_\mathbb{Z}(f(C_\mathbb{Z}(y))) \neq C_\mathbb{Z}(g(C_\mathbb{Z}(x)))$ evaluates to $\top$ and not to true. In fact $C_\mathbb{Z}(7C_\mathbb{Z}(x)^2 - 1) = C_\mathbb{Z}(7\mathbb{Z} - 1) = -1 + 7\mathbb{Z}$, while $C_\mathbb{Z}(x^2) = \mathbb{Z}$, and $-1 + 7\mathbb{Z} \neq \mathbb{Z}$ evaluates to $\top$. This happens because the square function cannot be precisely expressed as an element of the domain of congruences. Therefore $C_\mathbb{Z}$ is not complete for the function $g(x) = x^2$. The completeness refinement for this domain is given by $\mathit{gfp}(\lambda X.C_\mathbb{Z} \sqcap \mathcal{M}(g(X)))$, namely $\mathit{gfp}(\lambda X.C_\mathbb{Z} \sqcap \mathcal{M}(\{x^2 | x \in X\}))$, meaning that we add to the domain $C_\mathbb{Z}$ all the elements containing the squares of the elements in $C_\mathbb{Z}$. Let us consider the opaque predicate given by $7y^4 - 1 \neq x^4$. Also in this case we have that the abstract domain of congruences $C(\mathbb{Z})$ is not complete for the function $g(x) = x^4$, and therefore the domain of congruences is not precise for the evaluation of the opaque predicate. The completeness refinement that makes $C(\mathbb{Z})$ $\mathcal{F}$-complete w.r.t. g is $\mathit{gfp}(\lambda X.C_\mathbb{Z} \sqcap \mathcal{M}(g(X)))$, namely $\mathit{gfp}(\lambda X.C_\mathbb{Z} \sqcap \mathcal{M}(\{x^4 | x \in X\}))$. This means that to the domain $C(\mathbb{Z})$ we have to add the set of all elements that are 4-th power of elements in $\mathbb{Z}$, that are 16-th power and so on. By construction we have: $\mathit{gfp}(\lambda X.C_\mathbb{Z} \sqcap \mathcal{M}(\{x^2 | x \in X\})) \sqsubseteq \mathit{gfp}(\lambda X.C_\mathbb{Z} \sqcap \mathcal{M}(\{x^4 | x \in X\}))$, meaning that the obfuscator that inserts the first opaque predicate is more potent than the one that inserts the second opaque predicate w.r.t. the attacker $C_\mathbb{Z}$. This is justified by considering that the completeness refinement of $C(\mathbb{Z})$ in the case of $7y^4 - 1 \neq x^4$ is strictly more abstract than the one for $7y^2 - 1 \neq x^2$. The next step to do now is to understand how much more effort is necessary to break the first opaque predicate instead of breaking the second one in terms of dynamic testing.

V. Conclusions

In this paper we used abstract interpretation in order to specify control obfuscation algorithms. This provides both a systematic derivation of obfuscation algorithms as abstractions of semantics and a method for comparing the resilience of opaque predicates in the lattice of abstract domains. Our method assumes that an attacker has a constrained observation on program's behavior. This is specified by modeling attackers as abstractions of trace semantics. Decidable abstractions correspond to attackers performing static program analysis, while undecidable ones correspond to dynamic testing of predicates. The use of abstract interpretation however ensures that, when the abstraction is complete, the attacker is capable to remove obfuscation, i.e., the inserted opaque predicates. Lot of work is necessary in order to validate our theory in practice. While measuring the resilience of opaque predicates in the lattice of abstract domains may provide an absolute and domain-theoretical taxonomy of attackers and obfuscators, it would be interesting to investigate the true effort, in terms of dynamic testing, which is necessary to join static analysis in order to break opaque predicates. We believe that this is proportional to the missing information in the abstraction modeling the static analysis with respect to its complete refinement.

REFERENCES

[1] C. Thomborson C. Collberg. Breaking abstrcations and unstructural data structures. In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '98*), 1998.

[2] C. Thomborson C. Collberg. Watermarking, tamper-proofing, and obduscation-tools for software protection. *IEEE Trans. Software Eng.*, pages 735–746, 2002.

[3] D. Low C. Collberg, C. Thomborson. A taxionomy of obduscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.

[4] D. Low C. Collberg, C. Thomborson. Manifactiring cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (*POPL '98*). ACM Press, 1998.

[5] J. Knight C. Wang, J. Hill and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report CS-2000-12, Department of Computer Science, University of Virginia, 2000.

[6] Y. Caseau. Abstract interpretation of constraints on order-sorted domains. In V. Saraswat and K. Ueda, editors, *Proc. of the 1991 Internat. Logic Programming Symp.* (*ILPS '91*), pages 435–452. The MIT Press, Cambridge, Mass., 1991.

[7] S. Chandrasekharan and S. Debray. Deobfuscation: Improving reverse engineering of obfuscated code. Draft, 2005.

[8] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[9] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In B. Jonsson and J. Parrow, editors, *Proc. of the 5th Internat. Conf. on Concurrency Theory* (*CONCUR '94*), volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer-Verlag, 1994.

[10] M. Comini and G. Levi. An algebraic theory of observables. In M. Bruynooghe, editor, *Proc. of the 1994 Internat. Logic Programming Symp.* (*ILPS '94*), pages 172–186. The MIT Press, Cambridge, Mass., 1994.

[11] P. Cousot. Semantic foundations of program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 303–342. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[12] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[13] P. Cousot. Types as abstract interpretations (Invited Paper). In *Conference Record of the 24th ACM Symp. on Principles of Programming Languages* (*POPL '97*), pages 316–331. ACM Press, New York, 1997.

[14] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 2000.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.

[16] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.

[17] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM Symp. on Principles of Programming Languages* (*POPL '92*), pages 83–94. ACM Press, New York, 1992.

[18] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, New York, NY, 2002. ACM Press.

[19] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[20] R. Giacobazzi. "Optimal" collecting semantics for analysis in a hierarchy of logic program semantics. In C. Puech and R. Reischuk, editors, *Proc. of the 13th Internat. Symp. on Theoretical Aspects of Computer Science* (*STACS '96*), volume 1046 of *Lecture Notes in Computer Science*, pages 503–514. Springer-Verlag, Berlin, 1996.

[21] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th International Static Analysis Symposium, SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 2001.

[22] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.

[23] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12(2-3):75–90, 1999.

[24] P. Granger. Static analysis of arithmetical congruences. *Intern. J. Computer Math.*, 30:165–190, 1989.

[25] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods Syst. Des.*, 6:11–44, 1995.

[26] D. Low. Java control flow obfuscation. Master of science thesis, Department of Computer Science, The University of Auckland, 1998.

[27] B. Monsuez. System F and abstract interpretation. In A. Mycroft, editor, *Proc. of the 2nd Internat. Static Analysis Symp.* (*SAS '95*), volume 983 of *Lecture Notes in Computer Science*, pages 279–295. Springer-Verlag, Berlin, 1995.

[28] P. Ørbæk. Can you trust your data? In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of the 6th Internat. Joint Conf. CAAP/FASE, Theory and Practice of Software Devolpment* (*TAPSOFT '95*), volume 915 of *Lecture Notes in Computer Science*, pages 575–589. Springer-Verlag, 1995.

[29] P. Ørbæk and J. Palsberg. Trust in the lambda-calculus. *J. Funct. Program.*, 7(6):557–591, 1997.

[30] R. Paige. Future directions in program transformations. In *ACM SIGPLAN Not.*, volume 32, pages 94–97, 1997.

[31] D. Plaisted. Theorem proving with abstraction. *Artif. Intell.*, 16:47–108, 1981.

[32] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstrcat interpretation. In *Proceedings of the 32th International Colloquium on Automata, Languages and Progrmming (ICALP'05 - Track B)*, volume 3580, pages 1325–1336. Springer-Verlag, 2005. July 11-15, Lisboa, Portugal.

[33] M. Madou S. K. Udupa, S. Debray. Deobfuscation: Reverse engineering obfuscated code. Draft, 2005.

[34] A. Venet. Abstract interpretation of the π-calculus. In M. Dam, editor, *Proc. of the 5th LOMAPS Meeting on Analysis and Verification of High-Level Concurrent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, Berlin, 1996.