Semantics-based Code Obfuscation by Abstract Interpretation

MILA DALLA PREDA ROBERTO GIACOBAZZI
Dipartimento di Informatica, Università di Verona
Strada le Grazie 15, 37134 Verona, Italy
mila.dallapreda@univr.it roberto.giacobazzi@univr.it

Abstract

In recent years code obfuscation has attracted research interest as a promising technique for protecting secret properties of programs. The basic idea of code obfuscation is to transform programs in order to hide their sensitive information while preserving their functionality. One of the major drawbacks of code obfuscation is the lack of a rigorous theoretical framework that makes it difficult to formally analyze and certify the effectiveness of obfuscating techniques. We face this problem by providing a formal framework for code obfuscation based on abstract interpretation and program semantics. In particular, we show that what is hidden and what is preserved by an obfuscating transformation can be expressed as abstract interpretations of program semantics. Being able to specify what is masked and what is preserved by an obfuscation allows us to understand its potency, namely the amount of obscurity that the transformation adds to programs. In the proposed framework, obfuscation and attackers are modeled as approximations of program semantics and the lattice of abstract interpretations provides a formal tool for comparing obfuscations with respect to their potency. In particular, we prove that our framework provides an adequate setting to measure not only the potency of an obfuscation but also its resilience, i.e., the difficulty of undoing the obfuscation. We consider code obfuscation by opaque predicate insertion and we show how the degree of abstraction needed to disclose different opaque predicates allows us to compare their potency and resilience.
Keywords: Code Obfuscation, Abstract Interpretation, Program Semantics, Static Program Analysis.

1 Introduction

A major issue in computer security is the protection of proprietary software against *malicious host* attacks that usually aim at stealing, modifying or tampering with the code in order to obtain (economic) advantages over it. A key challenge in defending code that is running on an untrusted host is that there is no limit on the techniques that the host can use to extract sensitive data from the code and to violate its intellectual property and integrity. Malicious reverse-engineering, software piracy and software tampering are the most common malicious host attacks against proprietary programs [3]. Given a software application, the aim of reverse-engineering is to analyze it in order to understand its inner working. The information collected during the reverse-engineering process can be used either to improve the application, e.g., platform optimization and bug-fixes, or for unlawful purposes (so-called *malicious reverse-engineering*), e.g., identification

of vulnerabilities in binaries and unauthorized modifications such as bypassing password protection. Let us observe that both software tampering and software piracy need a preliminary reverse-engineering phase in order to understand the inner working of the program that they want to tamper with or to steal. Thus, preventing malicious reverse-engineering is a crucial issue when defending programs against malicious host attacks. *Code obfuscation* represents one of the most promising techniques to prevent malicious reverse-engineering of software. The idea is to transform programs in order to make them more difficult to understand and analyze while preserving their functionality.

The problem. According to a standard definition, an obfuscator is a *potent* program transformation that preserves the observational behaviour of programs, i.e., the input-output behaviour [5, 7, 8]. In this context, a transformation is potent when the transformed program is more complex, i.e., more difficult to reverse-engineer, than the original one. Consequently, the notion of code obfuscation is based on a fixed metric for program complexity, which is usually defined in terms of syntactic program features, such as code length, number of nesting levels and numbers of branching instructions [7]. To the best of our knowledge, there are no complexity measures based on program semantics, which we suggest may provide a deeper insight in the true potency of code obfuscation.

Many researchers recognise that one of the major drawbacks of code obfuscation is the lack of a rigorous theoretical background. In fact, the absence of a theoretical basis makes it difficult to formally analyze and certify the effectiveness of these techniques in contrasting malicious host attacks. In particular, it is hard to compare different obfuscating transformations with respect to their resilience to attacks and this makes it difficult to understand which technique is better to use in a given scenario. Little theoretical work on code obfuscation exists, and the design of a formal framework for modeling, studying and relating obfuscating transformations and attacks is still in an early stage.

The idea. In order to formalize and quantify the amount of "obscurity" added by an obfuscating transformation, namely how much more complex the transformed program is to reverse-engineer with respect to the original one, we need a formal model for obfuscation as well as for attack. Reverse-engineering typically consists of static and dynamic program analyses which can both be modeled as abstractions of program semantics. In fact, static program analysis can be specified as an abstract interpretation, i.e., as an approximation, of program semantics [14], while dynamic analysis can be seen as a possibly undecidable approximation of program semantics. Recall that program semantics formalizes program behaviour and that the precision of this description depends on the level of abstraction of the considered semantics, namely on the level of abstraction of the domain over which the semantics is computed. In particular, Cousot [13] defines a hierarchy of semantics, where semantics at different levels of abstractions are specified as successive approximations of a given concrete semantics. In the following, concrete program semantics refers to trace semantics, which observes step by step the history of each possible computation, while abstract semantics refers to any approximation of trace semantics. Note that the semantics modeling the input-output (observational) behaviour of a program is an element of this hierarchy because it is an abstraction of trace semantics.

Our idea is to provide a formal basis for code obfuscation by considering the effects that obfuscating transformations have on trace semantics and by modeling attacks as

abstractions of trace semantics. In order to reason about the semantic aspects of obfuscation we refer to the formal framework introduced by Cousot and Cousot [16], where the relation between syntactic and semantic transformations is formalized in terms of abstract interpretation by considering programs as abstractions of their semantics.

Main contribution. We provide a theoretical framework based on program semantics and abstract interpretation, in which we formalize, study and relate different obfuscating transformations with respect to their potency. It is worth remarking that our formal framework is language-independent, meaning that it can deal with the trace semantics of any programming language that can be specified as a transition system. Our examples will be instantiated in a simple imperative language.

As noticed above, attacks – static and dynamic analyzers – can be modeled as abstractions of trace semantics, where the abstract domain of computation modeling an attack precisely captures the amount of information that the attack is able to deduce while observing a program. Thus, a coarse abstraction models an attack that observes simple semantic properties, while finer abstractions, being closer to program trace semantics, model attacks that are interested in the details of computation. In this setting, an attack $\mathcal{A}$ is defeated by a program transformation $\mathbb{t}$, i.e., $\mathbb{t}$ is potent with respect to $\mathcal{A}$, when the semantic property modeling $\mathcal{A}$ is not preserved by $\mathbb{t}$. Following this observation we characterize the obfuscating behaviour of a transformation $\mathbb{t}$ in terms of the most concrete property $\delta_{\mathbb{t}}$ it preserves on program trace semantics. This allows us to provide a formalization of code obfuscation that is parametric on the most concrete semantic property it preserves. In particular, any transformation $\mathbb{t}$ can be seen as a $\delta_{\mathbb{t}}$-obfuscator that is potent with respect to any attack $\mathcal{A}$ finer than $\delta_{\mathbb{t}}$, and that preserves all the aspects of program behaviour that are expressed by $\delta_{\mathbb{t}}$. According to this formalization, any program transformation can be seen as a code obfuscation where the most concrete preserved property precisely expresses what can still be known after obfuscation, namely what it is possible to deduce of the original program from the analysis of the obfuscated one. In order to characterize the obfuscating behaviour of any given program transformation, we provide a systematic methodology for deriving the most concrete property preserved by a given transformation.

Since the semantic properties are modeled, as usual, by abstractions of trace semantics, we can compare different obfuscating transformations with respect to the degree of abstraction of the most concrete property they preserve. Given a δ-obfuscator, the more abstract δ is, the bigger is the set of attacks that it is able to defeat, which means that the transformation potency is high and many details of the original program behaviour have been lost during the obfuscation phase. On the other hand, when δ is close to trace semantics, it means that few details of the original program have been hidden by the obfuscation and that the transformation has a low potency.

The semantics-based definition of code obfuscation, together with the abstract interpretation-based model of attacks, turns out to be particularly useful when considering control code obfuscation by opaque predicate insertion. Here, the obfuscating transformation confuses the original control flow of programs by inserting "fake" conditional branches guarded by *opaque predicates*: predicates whose constant value is known by the obfuscator but which is difficult for an attacker to deduce. This confuses any attack that is not aware of the constant value of the inserted opaque predicate and erroneously sees both branches as possible (even if one is never executed at run time). Even if opaque predicate insertion does not significantly affect program trace semantics, since during execution the opaque predicate always evaluates the same, it might

considerably affect the abstract semantics computed on the abstract domain modeling the attack. In this case, we have that an attack is able to break opaque predicate insertion only if its abstract domain is precise enough to detect the opaqueness of the inserted predicates. In particular, modeling attacks as abstract domains allows us to prove that the degree of precision needed by an attack to break an opaque predicate can be expressed as a completeness problem in abstract interpretation. This result is particularly interesting because it provides a precise formalization of the amount of information needed by an attack to disclose a given opaque predicate. Moreover, we can measure the resilience of an opaque predicate with respect to an attacker $\mathcal{A}$ in terms of the amount of information that $\mathcal{A}$ needs in order to disclose the opaque predicate, and this allows us to compare the resilience of different opaque predicates with respect to $\mathcal{A}$.

1.1 Related work

Some early attempts at technical software protection, later called code obfuscation, are described in [24]. In recent years, code obfuscation has attracted the interest of researchers as a promising defence technique against malicious reverse-engineering of software, leading to the design of different obfuscating transformations (e.g., [3, 4, 5, 8, 30, 33, 40]). Collberg et al. present a number of obfuscating transformations classified according to the kind of information they target [7]. *Layout obfuscators* act on code information that is unnecessary to its execution. These transformations include the removal of comments and the change of identifiers. For example, by replacing identifiers of methods and variables with meaningless identifiers, any information on the functionality of a method or on the role of a variable is removed. *Data obfuscators* operate instead on program data structures. These transformations may for example alter how data are grouped together, making it more difficult for a reverse-engineer to restore the program's data structure. These transformations can split, fold or merge arrays in order to complicate the access to arrays, for example by transforming a two-dimensional array in a one-dimensional array and vice versa. Data obfuscators may also change how data are ordered. For example, they can reorder arrays using a function $f(i)$ to determine the position of the i-th element of the array, while the i-th element is usually stored in the i-th position of the array. *Control code obfuscators* attempt to confuse program control flow. These transformations often relay on the existence of opaque predicates, whose insertion allows to break the original control flow of a program.

Recall that the process of reverse-engineering an executable program typically begins with a disassembly phase, which translates machine code to assembly code, then is followed by a number of decompilation steps that try to recover high-level code from assembly code. Thus, in order to complicate reverse-engineering, we can either confuse the disassembly or the decompilation phase. Decompilation mainly involves static analysis of assembly code, including data-flow, control-flow and type analysis. Therefore, a program transformation that obstructs such static analyses acts as an obfuscating technique. Most of the existing obfuscating transformations, including the ones presented above, focus on the decompilation phase (e.g., [5, 7, 8, 40, 33]), while less attention is paid to obstructing disassembly. However, recently, some work has been done in the direction of obfuscating executable code in order to thwart well-known static disassembly techniques, such as linear sweep and recursive traversal [30]. Obstructing correct disassembly can be achieved also by changing repeatedly the program code while it executes [31].

Wang et al. observe that any intelligent tampering attack requires knowledge of the program semantics, usually obtained by static analysis [40]. Thus, they provide a code obfuscation technique based on variable aliasing that drastically reduces the precision of static analysis, because aliasing analysis is computationally hard. However, this approach is restricted to the case of intra-procedural analyses. A software obfuscation technique based on obstructing inter-procedural analysis and on the difficulty of alias analysis is proposed in [33], together with a theoretical proof of its effectiveness. Static analysis is conservative, meaning that the properties deduced by static deobfuscating techniques are weaker than the ones that may actually be true (this corresponds to an over-approximation). This guarantees soundness, although the inferred properties may be so weak to be useless. On the other side, a dynamic analysis precisely observes only a subset of all possible execution paths of a program (this corresponds to an under-approximation). Recent work on combining static and dynamic program analysis seems to provide a set of heuristics for disclosing some obfuscating techniques [39].

A well known negative theoretical result on code obfuscation is given by Barak et al. [2], who show that code obfuscation is impossible. This result seems to prevent code obfuscation entirely. However, this result is stated and proved in the context of a rather specific model of code obfuscation. Barak et al. [2] define an obfuscator as a program transformer $\mathcal{O}$ satisfying the following conditions: (1) $\mathcal{O}(P)$ is functionally equivalent to P, (2) the slowdown of $\mathcal{O}(P)$ with respect to P is polynomial both in time and space, and (3) anything that one can compute from $\mathcal{O}(P)$ can also be computed from the input-output behaviour of P. Hence, this formalizes an "ideal" obfuscator, where the original and obfuscated program have identical behaviour (1,2) and where the obfuscated program is unintelligible to an adversary (3). In practical contexts, these constraints can be relaxed. In particular, in [3, 7, 8, 33, 40] the authors consider a number of obfuscating transformations that make the obfuscated program significantly slower or larger than the unobfuscated program. These proposals even allow the obfuscated program to have different side effects than the original one, or not to terminate when the original program terminates with an error condition. The only requirement they make is that the *observable behaviour* — the behaviour observed by a generic user — of the two programs should be identical. Besides, many researchers are interested in transformations that raise the difficulty of reverse-engineering a program, even if they do not make it impossible as required by point (3) of the definition of Barak et al. In fact, protection can be guaranteed by a sufficiently difficult transformation that requires so many resources to be undone, as to make it uneconomical for an adversary to analyze the transformed program. Moreover, the "ideal" obfuscator of Barak et al. has to be able to protect *every* program. In fact the impossibility of code obfuscation is proved by providing a contrived class of functions that are not obfuscatable. It would be interesting to characterize the portion of programs of practical interest to which this negative result can be applied. By relaxing the constraint of Barak's definition, we are able to study the practical possibilities of obfuscating significant programs.

2 Preliminaries

2.1 Basic notions

Let S and T be two sets. Then $\wp(S)$ denotes the powerset of S, $S \setminus T$ denotes the set-difference between S and T, $S \subset T$ denotes strict inclusion and $S \subseteq T$ denotes

inclusion.

$\langle P, \leq \rangle$ denotes a poset P with ordering relation $\leq$, while $\langle P, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice P, with ordering $\leq$, least upper bound (*lub*) $\vee$, greatest lower bound (*glb*) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. Often, $\leq_P$ will be used to denote the underlying ordering of a poset P, and $\vee_P$, $\wedge_P$, $\top_P$ and $\bot_P$ denote the basic operations and elements of a complete lattice P. Given two ordered structures C and A the notation $C \cong A$ denotes that C and A are isomorphic. The downward closure of $S \subseteq P$ is $\downarrow S \stackrel{\text{def}}{=} \left\{ x \in P \,\middle|\, \exists y \in S. x \leq y \right\}$, and for $x \in P$, $\downarrow x$ is a short-land for $\downarrow \{x\}$, and the upward closure $\uparrow$ is dually defined.

We use the symbol $\sqsubseteq$ to denote pointwise ordering between functions: if X is any set, P is a poset and $f, g : X \to P$ then $f \sqsubseteq g$ if for all $x \in X, f(x) \leq g(x)$. If $f : S \to T$ and $g : T \to Q$ then $g \circ f : S \to Q$ denotes the composition of f and g, i.e., $g \circ f = \lambda x.g(f(x))$. If $f : C \to C$ is a unary function then the inverse image is defined as $f^{-1}(y) = \left\{ x \,\middle|\, f(x) = y \right\}$. A function $f : P \to Q$ on posets is (Scott)-continuous when f preserves *lub*'s of countable chains in P, while, dually, it is co-continuous when f preserves *glb*'s of countable chains in P. A mapping $f : C \to D$ on complete lattices is additive(resp. co-additive) when for any $Y \subseteq C, f(\vee_C Y) = \vee_D f(Y)$ (resp. $f(\wedge_C Y) = \wedge_D f(Y)$). The least and greatest fixpoint of an operator f on a poset $\langle P, \leq_P \rangle$, when they exist, are respectively denoted by $\mathit{lfp}^{\leq_P} f$ and $\mathit{gfp}^{\leq_P} f$, or by $\mathit{lfp}\, f$ and $\mathit{gfp}\, f$ when the partial order is clear from the context. The well-known Knaster-Tarski's theorem states that any continuous operator $f : C \to C$ on a complete lattice C admits a least fixpoint and the following characterization holds: $\mathit{lfp}^{\leq_C} f = \bigvee_{i \in \mathbb{N}} f^i(\bot_C)$, where for any $i \in \mathbb{N}$ and $x \in C$, the i-th power of f in x is inductively defined as follows: $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$. Dually, if f is co-continuous then $\mathit{gfp}^{\leq_C} f = \bigwedge_{i \in \mathbb{N}} f^i(\top_C)$.

2.2 Abstract interpretation

According to a widely recognized definition: *"Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems"* [12]. The key idea of abstract interpretation is that the behaviour of a program at different levels of abstraction is an approximation of its (concrete) semantics. The concrete program semantics is computed on the so-called *concrete domain*, i.e., the poset of mathematical objects on which the program runs, here denoted by $\langle C, \leq_C \rangle$ where the ordering relation encodes relative precision: $c_1 \leq_C c_2$ means that c_1 is a more precise (concrete) description than c_2. For instance, the concrete domain for a program with integer variables is simply given by the powerset of integer numbers ordered by subset inclusion $\langle \wp(\mathbb{Z}), \subseteq \rangle$. Approximation is encoded by an *abstract domain* $\langle A, \leq_A \rangle$, which is a poset of abstract values that represent some approximated properties of concrete objects. Also in the abstract domain, the ordering relation models relative precision: $a_1 \leq_A a_2$ means that a_1 is a better approximation (i.e., more precise) than a_2. For example, we may be interested in the sign of an integer variable, so that a simple abstract domain for this property may be $\mathit{Sign} = \{\top, -, 0, +, \bot\}$ where $\top$ gives no sign information, while the meaning of $-$(resp. 0)(resp. $+$) is that the value of a defined variable is negative(resp. null)(resp. positive), and $\bot$ represent an uninitialized variable or an error (e.g., division by zero). Thus, we have that $\bot < -, 0, + < \top$, so that, in particular, the abstract values $-$, 0 and $+$ are incomparable.

Galois connections. In standard abstract interpretation, concrete and abstract domains are related through a Galois connection (GC), i.e., an adjunction [14, 15]. With the two equivalent notations (C, α, γ, A) and $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ we denote a GC where the concrete domain C is related to the abstract domain A by the abstraction map $\alpha : C \to A$ and the concretization map $\gamma : A \to C$ that give rise to an adjunction: $\forall a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Thus, $\alpha(c) \leq_A a$ and, equivalently, $c \leq_C \gamma(a)$ means that a is a sound approximation in A of c. GCs ensure that $\alpha(c)$ actually provides the best possible approximation in the abstract domain A of the concrete value $c \in C$. In the abstract domain $Sign$, for example we have that $\alpha(\{-1, -5\}) = -$ while $\alpha(\{-1, +1\}) = \top$.

Recall that a tuple (C, α, γ, A) is a GC iff α is additive iff γ is co-additive. This means that whenever we have an additive(resp. co-additive) function f between two domains we can always build a GC by considering the right(resp. left) adjoint map induced by f. In fact, every abstraction map induces a concretization map and vice versa, formally $\gamma(y) = \bigvee \{\ x \mid\ \alpha(x) \leq y\ \}$ and $\alpha(x) = \bigwedge \{\ y \mid\ x \leq \gamma(y)\ \}$.

When a GC is such that $\alpha \circ \gamma = \lambda x.x$, we have a *Galois insertion* (GI) denoted $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\!\!\rightarrow}} A$. Any GC may be lifted to a GI by identifying, in an equivalence class, those values of the abstract domain with the same concretization.

Of course, abstract domains can be compared with respect to their relative degree of precision: if A_1 and A_2 are both abstract domains of a common concrete domain C, A_1 is more precise than A_2, denoted by $A_1 \sqsubseteq A_2$, when for any $a_2 \in A_2$ there exists $a_1 \in A_1$ such that $\gamma_1(a_1) = \gamma_2(a_2)$, i.e., when $\gamma_2(A_2) \subseteq \gamma_1(A_1)$. For example, the well-known abstract domain of integer intervals is obviously more precise than the sign abstract domain $Sign$.

Upper closure operators. Abstract interpretation can be equivalently formalized in terms of *upper closure operators* instead of Galois connections [15]. The two approaches are equivalent, modulo isomorphic representations of the domain object. An upper closure operator, or closure, on poset $\langle C, \leq_C \rangle$ is an operator $\varphi : C \to C$ that is monotone, idempotent and extensive (i.e., $\forall x \in C : x \leq_C \varphi(x)$). Let us recall that each closure φ is uniquely determined by the set of its fixpoints, which is its image $\varphi(C)$. Moreover, a subset $X \subseteq C$ is a set of fixpoints of a closure iff X is a *Moore family* of C, i.e., $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\ \wedge S \mid\ S \subseteq X\ \}$, where $\wedge\emptyset = \top \in \mathcal{M}(X)$. For any $X \subseteq C$, $\mathcal{M}(X)$ is called the *Moore closure* of X in C, i.e., $\mathcal{M}(X)$ is the least (with respect to subset inclusion) subset of C which contains X and which is a Moore family of C. Often, we will identify closures with their sets of fixpoints. If (C, α, γ, A) is a GC then $\varphi \stackrel{\text{def}}{=} \gamma \circ \alpha$ is the closure associated with A, such that $\varphi(C)$ is a complete lattice isomorphic to A, i.e., $\varphi(C) \cong A$. Given a GC (C, α, γ, A), the closure $\gamma \circ \alpha$ associated to the abstract domain A can be thought of as the "logical meaning" of A in C, since this is shared by any other abstract representation for the objects of A. Thus, the closure operator approach is convenient when reasoning about properties of abstract domains independently from the representation of their objects.

Lattice of abstract interpretation. The ordered set $\langle uco(C), \sqsubseteq \rangle$ of all upper closure operators of C, plays the role of the lattice of abstract interpretations of C [14, 15]. Let $\varphi_i(C) \cong A_i$, the pointwise ordering on $uco(C)$ corresponds precisely to the standard ordering used to compare abstract domains with regard to their precision: A_1 is more precise than A_2, i.e., $A_1 \sqsubseteq A_2$, iff $\varphi_1 \sqsubseteq \varphi_2$ in $uco(C)$ iff $\varphi_2(C) \subseteq \varphi_1(C)$. Let $\{A_i\}_{i \in I} \subseteq uco(C)$: $\sqcup_{i \in I} A_i$ is the *least* (with respect to $\sqsubseteq$) *common abstraction* of

all the A_i's, i.e., the most concrete domain in $uco(C)$ which is abstraction of all A_i's. Moreover $\sqcap_{i\in I} A_i$ is the *reduced product* of all the A_i's, i.e., the most abstract domain in $uco(C)$, which is more concrete than every A_i's. *Complementation* corresponds to the inverse of reduced product [10], namely an operator that, given two domains $C \sqsubseteq D$, gives as result the most abstract domain $C \ominus D$, whose reduced product with D is exactly C, i.e., $(C \ominus D) \sqcap D = C$. Therefore we have that $C \ominus D \stackrel{\text{def}}{=} \sqcup \{E \in uco(C) | D \sqcap E = C\}$.

Soundness and completeness of abstract functions. In abstract interpretation, a concrete semantic operation is then formalized as any (possibly n-ary) function $f : C \to C$ on the concrete domain. For example, a (unary) integer squaring operation sq on the concrete domain $\wp(\mathbb{Z})$ is given by $sq(X) = \{x^2 \in \mathbb{Z} \mid x \in X\}$, while an integer increment (by one) operation $succ$ is given by $succ(X) = \{x + 1 \in \mathbb{Z} \mid x \in X\}$. A concrete semantic operation must be approximated on some abstract domain A by a *sound abstract operation* $f^\sharp : A \to A$. This means that $f^\sharp$ must be a correct approximation of f in A: for any $c \in C$ and $a \in A$, if a approximates c then $f^\sharp(a)$ must approximate $f(c)$. This is therefore encoded by the condition: for all $c \in C$, $\alpha(f(c)) \leq_A f^\sharp(\alpha(c))$. For example, a correct approximation $sq^\sharp$ of sq on the abstract domain $Sign$ can be defined as follows: $sq^\sharp(\bot) = \bot$, $sq^\sharp(0) = 0$, $sq^\sharp(-) = +$, $sq^\sharp(+) = +$ and $sq^\sharp(\top) = \top$; while a correct approximation $succ^\sharp$ of $succ$ on $Sign$ is given by: $succ^\sharp(\bot) = \bot$, $succ^\sharp(-) = \top$, $succ^\sharp(0) = +$, $succ^\sharp(+) = +$ and $succ^\sharp(\top) = \top$. Soundness can be also equivalently stated in terms of the concretization map, i.e., for all $a \in A$, $f(\gamma(a)) \leq_C \gamma(f^\sharp(a))$. These two equivalent soundness conditions can be strengthened to two different (i.e., incomparable) notions of *completeness*. When $\alpha \circ f = f^\sharp \circ \alpha$ holds, the abstract function $f^\sharp$ is said to be *backward-complete* for f. On the other hand, when $f \circ \gamma = \gamma \circ f^\sharp$ holds, $f^\sharp$ is *forward-complete* for f. Both backward($\mathcal{B}$) and forward($\mathcal{F}$)-completeness encode an ideal situation where no loss of precision arise in abstract computations: $\mathcal{B}$-completeness considers abstractions on the output of operations while $\mathcal{F}$-completeness considers abstractions on the input to operations. For example, $sq^\sharp$ is $\mathcal{B}$-complete for sq on $Sign$ while it is not $\mathcal{F}$-complete because $sq(\gamma(+)) = \{x^2 \in \mathbb{Z} \mid x > 0\} \subsetneq \{x \in \mathbb{Z} \mid x > 0\} = \gamma(sq^\sharp(+))$. Also, observe that $succ^\sharp$ is neither backward nor forward complete for $succ$ on $Sign$. The two notions of completeness can be expressed in terms of closure operators, in particular, $\varphi \in uco(\wp(C))$ is $\mathcal{B}$-complete for f if $\varphi \circ f = \varphi \circ f \circ \varphi$, while it is $\mathcal{F}$-complete for f when $f \circ \varphi = \varphi \circ f \circ \varphi$. While any abstract domain A induces the so-called canonical *best correct approximation* $f^A \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \to A$ of $f : C \to C$ in A, not all abstract domains induce a $\mathcal{B}(\mathcal{F})$-complete abstraction. It turns out that both $\mathcal{B}$ and $\mathcal{F}$-completeness are abstract domain properties, namely they only depend on the structure of the underlying abstract domain in the sense that the abstract domain A determines whether it is possible to define a backward or forward complete operation $f^\sharp$ on A [22, 23]. The following result gives the basis for the definition of a systematic method for minimally refining a domain in order to make it complete for a given function.

Theorem 1 *[22, 23] Let $f : C \to C$ be continuous and $\varphi \in uco(C)$. Then:*

- *φ is $\mathcal{B}$-complete for f iff* $\bigcup_{y \in \varphi(C)} max(f^{-1}(\downarrow y)) \subseteq \varphi(C)$
- *φ is $\mathcal{F}$-complete for f iff* $\forall x \in \varphi(C).f(x) \in \varphi(C)$

This means that $\mathcal{B}$-complete domains are closed under maximal inverse image of the function f, while $\mathcal{F}$-complete domains are closed under direct image of f. Let us recall the definition of completeness refinement operators $\mathcal{R}_f^{\mathcal{B}}$ and $\mathcal{R}_f^{\mathcal{F}}$.

Definition 1 *[22] Let C be a complete lattice and $f : C \to C$ be a continuous function. We define $\mathcal{R}_f^{\mathcal{B}}, \mathcal{R}_f^{\mathcal{F}} : uco(C) \to uco(C)$ such that:*

- $\mathcal{R}_f^{\mathcal{B}} = \lambda X \in uco(C).\mathcal{M}(\bigcup_{y \in X} max(f^{-1}(y)))$;
- $\mathcal{R}_f^{\mathcal{F}} = \lambda X \in uco(C).\mathcal{M}(f(X))$.

Thus, given a continuous function $f : C \to C$ and an abstract domain $A \in uco(C)$, the more abstract domain which includes A and is $\mathcal{B}$-complete for f is $gfp(\lambda X.A \sqcap \mathcal{R}_f^{\mathcal{B}}(X))$, while the more abstract domain which includes A and is $\mathcal{F}$-complete for f is $gfp(\lambda X.A \sqcap \mathcal{R}_f^{\mathcal{F}}(X))$ [22, 23].

Abstract semantics. As observed earlier, one interest of abstract interpretation theory is the systematic design of approximate semantics of programs. Let us consider the concrete semantics $S[\![P]\!]$ of program P given, as usual, in fixpoint form $S[\![P]\!] = lfp^{\sqsubseteq} F[\![P]\!]$, where the semantic transformer $F[\![P]\!]$ is monotone an defined on the concrete domain of objects C. Given a GC (C, α, γ, A), the abstract semantics $S^A[\![P]\!]$ can be chosen as $lfp^{\leq_A} F^A[\![P]\!]$, where $F^A = \alpha \circ F \circ \gamma$ is given by the best correct approximation of F in A. The following well known result (see e.g., [15]) states that if the abstract domain A is $\mathcal{B}$-complete for a monotone function $f : C \to C$, then $lfp^{\leq_A} f^A = \alpha(lfp^{\leq_C} f)$.

Theorem 2 [FIXPOINT TRANSFER] *Given a GC (C, α, γ, A), and a concrete monotone function $f : C \to C$, if $\alpha \circ f = f^A \circ \alpha$ (resp. $\alpha \circ f \leq_A f^A \circ \alpha$) then $\alpha(lfp^{\leq_C} f) = lfp^{\leq_A} f^A$ (resp. $\alpha(lfp^{\leq_C} f) \leq_A lfp^{\leq_A} f^A$).*

This means that if the abstract domain is $\mathcal{B}$-complete for the semantic transfer F, then the abstract semantics coincides with the abstraction of the concrete semantics, i.e., $S^A[\![P]\!] = \alpha(S[\![P]\!])$.

2.3 Syntactic and semantic program transformations

A program transformation is a meaning-preserving mapping defined on programming languages [35], whose aims are, for example, to improve the reliability, the productivity, the maintenance, the security, or the analysis of code. Commonly used program transformations include constant propagation [29], partial evaluation [9, 28], slicing [41], reverse-engineering [42], compilation [37], code obfuscation [8] and software watermarking [6]. In [16] Cousot & Cousot formally define the relation between syntactic and semantic program transformations in terms of abstract interpretation. In particular, the authors provide a language-independent methodology for systematically deriving syntactic program transformations as approximations of the semantic ones (for which it is easier to prove meaning preservation).

In the following, syntactic arguments are between double square brackets $[\![...]\!]$ while semantic and mathematical arguments are between round brackets $(...)$. Given the set $\mathbb{P}$ of all possible programs, let $S[\![P]\!] \in \mathcal{D}$ denote the semantics of program $P \in \mathbb{P}$. The semantic domain $\mathcal{D}$ is a poset $\langle \mathcal{D}, \sqsubseteq \rangle$, where the partial order $\sqsubseteq$ denotes

relative precision, i.e., $\mathcal{Q} \sqsubseteq \mathcal{S}$ means that semantics $\mathcal{S}$ contains less information than semantics $\mathcal{Q}$. The semantic ordering $\sqsubseteq$ induces an order $\preccurlyeq$ on the domain $\mathbb{P}$ of programs, where $P \preccurlyeq Q \stackrel{\text{def}}{=} (S[\![P]\!] \sqsubseteq S[\![Q]\!])$. Thus, $\langle \mathbb{P}/_{=\!=}, \preccurlyeq \rangle$ is a poset and $\mathbb{P}/_{=\!=}$ denotes the classes of syntactically equivalent programs, where $P =\!= Q \stackrel{\text{def}}{=} (S[\![P]\!] = S[\![Q]\!])$.

According to Cousot [16], we denote with $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ a syntactic program transformation and with $t : \mathcal{D} \to \mathcal{D}$ its semantic counterpart that, given the semantics $S[\![P]\!]$ of program P, returns the semantics $S[\![\mathbb{t}[\![P]\!]]\!]$ of the syntactically transformed program. A program transformation $\mathbb{t}$ is *correct* if it is meaning preserving with respect to some *observational abstraction* $\alpha_{\mathcal{O}}$, namely if $\forall P \in \mathbb{P} : \ \alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(S[\![\mathbb{t}[\![P]\!]]\!])$, where $\alpha_{\mathcal{O}}$ could be, for example, the observation of the input-output behaviour of programs. Considering programs as abstractions of their semantics leads to the following Galois insertion:

$$\langle \mathcal{D}, \sqsubseteq \rangle \underset{\mathbb{p}}{\overset{S}{\leftrightarrows}} \langle \mathbb{P}/_{=\!=}, \preccurlyeq \rangle \tag{1}$$

where $\mathbb{p}[\![\mathcal{S}]\!]$ is the simplest program whose semantics upper approximates $\mathcal{S} \in \mathcal{D}$. Observe that (1) is a Galois insertion thanks to the fact that programs are considered up to syntactic equivalence. In fact, given a program $P \in \mathbb{P}$, $\mathbb{p}(S[\![P]\!]) =\!= P$ but potentially $\mathbb{p}(S[\![P]\!])$ may be different from P because of dead code elimination. Thus, $\mathbb{p}(S[\![P]\!])$ and P are syntactically equivalent since they differ only in the potential presence of dead code that does not appear in the semantics.

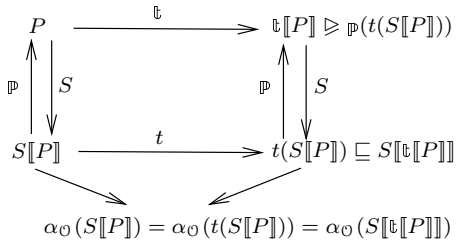

Figure 1: Syntactic-Semantic Program Transformations

The scheme in Figure 1 shows that each semantic transformation induces a syntactic transformation and vice versa:

$$t(S[\![P]\!]) \stackrel{\text{def}}{=} S[\![\mathbb{t}[\![\mathbb{p}(S[\![P]\!])]\!]]\!] \qquad\qquad \mathbb{t}[\![P]\!] \stackrel{\text{def}}{=} \mathbb{p}(t(S[\![P]\!]))$$

In particular, the above equation on the right expresses a syntactic transformation as an abstraction of the corresponding semantic transformation. In the following, we show how from this formalization it is possible to derive a systematic methodology for the design of syntactic transformations from the corresponding semantic ones. Observe that when the semantic transformation t relies on results of undecidable problems, any effective algorithm $\mathbb{t}$ is an approximation of the ideal transformation $\mathbb{p} \circ t \circ S$. This means that, in general, $\mathbb{p}(t(S[\![P]\!])) \preccurlyeq \mathbb{t}[\![P]\!]$ and from Galois insertion (1) this is equivalent to $t(S[\![P]\!]) \sqsubseteq S[\![\mathbb{t}[\![P]\!]]\!]$.

Any program transformation results, in general, in a loss of information on program semantics [16]. This approximation can be formalized in terms of the following Galois connection: $\langle \mathcal{D}, \sqsubseteq \rangle \underset{t}{\overset{\gamma_t}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq \rangle$, that composed with Galois connection (1) gives rise to the Galois connection: $\langle \mathbb{P}/_{=\!=}, \preccurlyeq \rangle \underset{\mathbb{t}}{\overset{\gamma_{\mathbb{t}}}{\leftrightarrows}} \langle \mathbb{P}/_{=\!=}, \preccurlyeq \rangle$. This means that, in general, syntactic and semantic transformations can both be seen as abstractions. Following this

observation, let us elucidate the steps that lead to the systematic designs of the syntactic transformation $\mathbb{t} \stackrel{\text{def}}{=} \mathbb{p} \circ t \circ S$ starting from the semantic transformation t:

Step 1 $\mathbb{p}(t(S[\![P]\!])) = \mathbb{p}(t(\mathit{lfp}F[\![P]\!]))$ where the semantics is expressed in fixpoint form: $S[\![P]\!] = \mathit{lfp}F[\![P]\!]$

Step 2 $\mathbb{p}(t(\mathit{lfp}F[\![P]\!])) = \mathbb{p}(\mathit{lfp}\hat{F}[\![P]\!])$ where $\hat{F} \stackrel{\text{def}}{=} t \circ F \circ \gamma_t$ follows from Theorem 2 with abstraction t, i.e., $t(\mathit{lfp}F[\![P]\!]) = \mathit{lfp}(t \circ F \circ \gamma_t)[\![P]\!]$ (resp. $\sqsubseteq$ for approximations)

Step 3 $\mathbb{p}(\mathit{lfp}\hat{F}[\![P]\!]) = \mathit{lfp}\mathbb{F}[\![P]\!]$ where $\mathbb{F} \stackrel{\text{def}}{=} \mathbb{p} \circ \hat{F} \circ S$ follows from Theorem 2 with abstraction $\mathbb{p}$, i.e., $\mathbb{p}(\mathit{lfp}\hat{F}[\![P]\!]) = \mathit{lfp}(\mathbb{p} \circ F \circ S)[\![P]\!]$

Step 4 $\mathbb{t}[\![P]\!] \stackrel{\text{def}}{=} \mathit{lfp}\mathbb{F}[\![P]\!]$ (resp. $\trianglelefteqslant$ for approximations)

Given the fixpoint formalization $\mathit{lfp}\mathbb{F}[\![P]\!]$ of the syntactic transformation, it is then possible to design an iterative algorithm on posets satisfying the ascending chain condition.

Algorithmic transformations. Let us say that a semantic transformation $t : \mathcal{D} \to \mathcal{D}$ is *algorithmic* if it is induced by a syntactic transformation $\mathbb{t}$, i.e., $t = S \circ \mathbb{t} \circ \mathbb{p}$, that is, if there exists an algorithm whose effects on program semantics are exactly the ones of transformation t.

Definition 2 *A semantic transformation $t : \mathcal{D} \to \mathcal{D}$ is algorithmic if there exists an algorithm, i.e., a syntactic transformation, $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ such that: $t = S \circ \mathbb{t} \circ \mathbb{p}$.*

In the following result we observe that the abstract domain $\mathbb{P}$ of programs is $\mathcal{F}$-complete for every concrete (semantic) algorithmic transformation t.

Lemma 1 *Considering the Galois insertion $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\mathbb{p}}{\overset{S}{\leftrightarrows}} \langle \mathbb{P}/_{=}, \trianglelefteqslant \rangle$ we have that the abstract domain $\mathbb{P}$ is $\mathcal{F}$-complete for every algorithmic transformation t.*

PROOF: Given an algorithmic transformation t, we have to show that $S \circ \mathbb{p} \circ t \circ S \circ \mathbb{p} = t \circ S \circ \mathbb{p}$. Let $\mathcal{X} \in \mathcal{D}$:

$$\begin{array}{lll} S[\![\mathbb{p}(t(S[\![\mathbb{p}(\mathcal{X})]\!]))]\!] & = S[\![\mathbb{p}(S[\![\mathbb{t}[\![\mathbb{p}(S[\![\mathbb{p}(\mathcal{X})]\!])]\!]]\!])]\!] & [t = S \circ \mathbb{t} \circ \mathbb{p},\ t \text{ algorithmic}] \\ & = S[\![\mathbb{t}[\![\mathbb{p}(S[\![\mathbb{p}(\mathcal{X})]\!])]\!]]\!] & [\mathbb{p} \circ S = id] \\ & = t(S[\![\mathbb{p}(\mathcal{X})]\!]) & [S \circ \mathbb{t} \circ \mathbb{p} = t] \end{array}$$

□

In particular, observe that $\mathcal{F}$-completeness means that $t \circ S = S \circ \mathbb{t}$, namely that there is no loss of precision between the semantic and syntactic transformations when we compare them on the concrete domain $\mathcal{D}$ of program semantics. This also implies that $\mathbb{t} = \mathbb{p} \circ t \circ S$. Thus, when considering algorithmic semantic transformations, the schema in Figure 1 commutes.

In this work we are interested in the study of the semantic counterpart of existing obfuscators and these semantic transformations are clearly algorithmic, since code obfuscation is, in general, an automatic program transformation. This means that there is no loss of precision between the semantic and the syntactic specification of an obfuscation. Moreover, given the semantic characterization t of an obfuscator the systematic methodology proposed by Cousot and Cousot [16] and reported at the end of Section 2.3, returns precisely the corresponding obfuscating algorithm $\mathbb{t} = \mathbb{p} \circ t \circ S$.

Syntactic Categories:

$\mathsf{n} \in \mathbb{Z}$	(integers)
$X \in \mathbb{X}$	(variable names)
$L \in \mathbb{L}_\jmath$	(labels)
$E \in \mathbb{E}$	(integer expressions)
$B \in \mathbb{B}$	(Boolean expressions)
$A \in \mathbb{A}$	(actions)
$C \in \mathbb{C}$	(commands)
$P \in \mathbb{P}$	(programs)

Value Domains:

$\mathcal{B}_\jmath = \{true, false\} \cup \{\jmath\}$	(truth values)
$n \in \mathbb{Z}$	(integers)
$\mathfrak{D}_\jmath = \mathfrak{D} \cup \{\jmath\}$	(variable values)
$\rho \in \mathfrak{E} = \mathbb{X} \to \mathfrak{D}_\jmath$	(environments)
$\Sigma = \mathbb{C} \times \mathfrak{E}$	(program states)

SYNTAX

$$E ::= \mathsf{n} \mid X \mid E_1 - E_2$$
$$B ::= \mathtt{true} \mid \mathtt{false} \mid E_1 < E_2 \mid \neg B_1 \mid B_1 \vee B_2$$
$$A ::= X := E \mid X :=? \mid B$$
$$C ::= L : A \to L'$$
$$\mathbb{P} ::= \wp(\mathbb{C})$$

SEMANTICS

Arithmetic Expr. $\mathbf{E} : \mathbb{E} \times \mathfrak{E} \to \mathfrak{D}_\jmath$

$$\mathbf{E}[\![\mathsf{n}]\!]\rho \stackrel{\text{def}}{=} n$$
$$\mathbf{E}[\![X]\!]\rho \stackrel{\text{def}}{=} \rho(X)$$
$$\mathbf{E}[\![E_1 - E_2]\!]\rho \stackrel{\text{def}}{=} \mathbf{E}[\![E_1]\!]\rho - \mathbf{E}[\![E_2]\!]\rho$$

Boolean Expr. $\mathbf{B} : \mathbb{B} \times \mathfrak{E} \to \mathcal{B}_\jmath$

$$\mathbf{B}[\![\mathtt{true}]\!]\rho \stackrel{\text{def}}{=} true$$
$$\mathbf{B}[\![\mathtt{false}]\!]\rho \stackrel{\text{def}}{=} false$$
$$\mathbf{B}[\![E_1 < E_2]\!]\rho \stackrel{\text{def}}{=} \mathbf{E}[\![E_1]\!]\rho < \mathbf{E}[\![E_2]\!]\rho$$
$$\mathbf{B}[\![\neg B]\!]\rho = \neg \mathbf{B}[\![B]\!]\rho$$
$$\mathbf{B}[\![B_1 \vee B_2]\!]\rho \stackrel{\text{def}}{=} \mathbf{B}[\![B_1]\!]\rho \vee \mathbf{B}[\![B_2]\!]\rho$$

Program Actions: $\mathbf{A} : \mathbb{A} \times \mathfrak{E} \to \wp(\mathfrak{E})$

$$\mathbf{A}[\![\mathtt{true}]\!]\rho \stackrel{\text{def}}{=} \{\rho\}$$
$$\mathbf{A}[\![X := E]\!]\rho \stackrel{\text{def}}{=} \{\rho[X := \mathbf{A}[\![E]\!]]\}$$
$$\mathbf{A}[\![X :=?]\!]\rho \stackrel{\text{def}}{=} \{\ \rho' \mid \exists z \in \mathbb{Z} : \rho' = \rho[X := z]\ \}$$
$$\mathbf{A}[\![B]\!]\rho \stackrel{\text{def}}{=} \{\ \rho' \mid \mathbf{B}[\![B]\!]\rho' = \mathtt{true} \wedge \rho' = \rho\ \}$$

Figure 2: A simple programming language [16].

2.4 The programming language

In the following we refer to the simple imperative language introduced in [16] whose syntax and semantics are reported in Figure 2. Given a set S, we use $S_\jmath$ to denote the set $S \cup \{\jmath\}$, where $\jmath$ represents an undefined value.[1] Commands can be either conditional or unconditional. A conditional command at label L is of the form $L : B \to L'$, where B is a boolean expression and L' is the label of the command to execute when B evaluates to $true$. An unconditional command at label L is of the form $L : A \to L'$, where A is an action and L' is the label of the command to be executed next. An action can be either an assignment $X := E$ or a random assignment $X :=?$ to variable

[1] We abuse notation and use $\jmath$ to denote undefined values of different types, since the type of the undefined value is usually clear from the context.

X, where A is an arithmetic expression and $?$ denotes a random value. Since each command explicitly mentions its successors, a program does not need to maintain an explicit sequence of commands and it can simply be specified as a set of commands, i.e., $\mathbb{P} = \wp(\mathbb{C})$. The `stop` command is expressed by $L : \texttt{stop} \eqsim L : \texttt{skip} \rightarrow \iota$, and the `skip` command by $L : \texttt{skip} \rightarrow L' \eqsim L : \texttt{true} \rightarrow L'$. The following example shows a program that computes the factorial which is written in the proposed programming language:

$$\begin{array}{ll} a:\ X :=? \ \rightarrow\ b & d : \texttt{stop} \\ b:\ F := 1 \ \rightarrow\ c & e:\ F := F * X \ \rightarrow\ f \\ c:\ (X = 1) \ \rightarrow\ d & f:\ X := X - 1 \ \rightarrow\ c \\ c:\ \neg(X = 1) \ \rightarrow e & \end{array}$$

In the following we report some auxiliary functions that allow us to isolate the labels and variables of a command or a program and that will be useful in the definition of the semantics of the language:

$$\begin{array}{ll} \mathit{lab}[\![L : A \rightarrow L']\!] \stackrel{\text{def}}{=} L & \mathit{lab}[\![P]\!] \stackrel{\text{def}}{=} \cup_{C \in P}\, \mathit{lab}[\![C]\!] \\ \mathit{var}[\![L : A \rightarrow L']\!] \stackrel{\text{def}}{=} \mathit{var}[\![A]\!] & \mathit{var}[\![P]\!] \stackrel{\text{def}}{=} \cup_{C \in P}\, \mathit{var}[\![C]\!] \\ \mathit{suc}[\![L : A \rightarrow L']\!] \stackrel{\text{def}}{=} L' & \mathit{act}[\![L : A \rightarrow L']\!] \stackrel{\text{def}}{=} A \end{array}$$

Let $\mathbb{L}_{\mathfrak{s}}$ be the set of program labels, let $\mathfrak{D}_{\mathfrak{s}}$ be the semantic domain of variables values, and let $\mathit{var}[\![A]\!]$ be the set of variables occurring in action A. An *environment* $\rho \in \mathfrak{E}$ maps each variable $X \in \mathit{dom}(\rho)$ to its value $\rho(X) \in \mathfrak{D}_{\mathfrak{s}}$. Given $V \subseteq \mathbb{X}$, we denote with $\rho|_V$ the restriction of environment ρ to the domain $\mathit{dom}(\rho) \cap V$, and with $\rho \setminus V$ the restriction of environment ρ to the domain $\mathit{dom}(\rho) \setminus V$. The notation $\rho[X := n]$ refers to environment ρ where value n is assigned to variable X. Let $\mathfrak{E}[\![P]\!]$ denote the set of environments of program P, namely of those environments whose domain is given by the set of program variables, i.e., $\mathit{dom}(\rho) = \mathit{var}[\![P]\!]$.

A *program state* is a pair $\langle \rho, C \rangle$, where C is the command that has to be executed in environment ρ. Let $\Sigma \stackrel{\text{def}}{=} \mathfrak{E} \times \mathbb{C}$ denote the set of all possible states, and $\Sigma[\![P]\!] \stackrel{\text{def}}{=} \mathfrak{E}[\![P]\!] \times \mathbb{C}$ the set of states of program P. As usual, the *transition relation* $\mathbf{C} : \Sigma \rightarrow \wp(\Sigma)$ between states specifies the set of states that are reachable from a given state. Thus, $\mathbf{C}(\langle \rho, C \rangle)$ returns the set of states that might be reached when executing command C in the environment ρ, formally:

$$\mathbf{C}(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \left\{\ \langle \rho', C' \rangle \ \middle|\ \rho' \in \mathbf{A}[\![\mathit{act}(C)]\!]\rho,\ \mathit{suc}[\![C]\!] = \mathit{lab}[\![C']\!]\ \right\}$$

A state $\langle \rho, C \rangle$ is a final/blocking state when $\mathbf{C}(\langle \rho, C \rangle) = \emptyset$. The transition relation between states can be specified with respect to a program P, $\mathbf{C}[\![P]\!] : \Sigma[\![P]\!] \rightarrow \wp(\Sigma[\![P]\!])$:

$$\mathbf{C}[\![P]\!](\langle \rho, C \rangle) \stackrel{\text{def}}{=} \left\{\ \langle \rho', C' \rangle \in \mathbf{C}(\langle \rho, C \rangle) \ \middle|\ \rho, \rho' \in \mathfrak{E}[\![P]\!] \wedge C' \in P\ \right\}$$

As usual, let Σ^+ denote the set of all possible finite nonempty sequences of states, Σ^ω the set of all infinite sequences of states, and $\Sigma^\infty \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$. Given a sequence of states $\sigma \in \Sigma^\infty$, let $|\sigma| \in \mathbb{N} \cup \{\omega\}$ denote its length, σ_i its i-th element and σ_f its final state when $\sigma \in \Sigma^+$. A *finite maximal execution trace* $\sigma \in S^n[\![P]\!]$ of program P is a finite sequence $\sigma_0 ... \sigma_{n-1} \in \Sigma^+$ of states of length n, i.e., $|\sigma| = n$, such that each state σ_i with $i \in [1, n-1]$ is a possible successor of the previous state σ_{i-1}, i.e., $\sigma_i \in \mathbf{C}(\sigma_{i-1})$, and the last state σ_{i-1} is a blocking state. Let $\mathfrak{T}[\![P]\!]$ denote the set of final/blocking states of program P, i.e., $\mathfrak{T}[\![P]\!] = \{\langle \rho, C \rangle \in \Sigma[\![P]\!] \mid \mathbf{C}(\langle \rho, C \rangle) = \emptyset\}$. The *maximal finite trace semantics* $S^+[\![P]\!]$ of program P is $S^+[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{n>0} S^n[\![P]\!]$

and it can be computed as $lfp^{\subseteq} F^+[\![P]\!]$, where $F^+[\![P]\!] : \wp(\Sigma^+[\![P]\!]) \to \wp(\Sigma^+[\![P]\!])$ is defined as:

$$F^+[\![P]\!](\mathcal{X}) \stackrel{\text{def}}{=} \mathfrak{T}[\![P]\!] \cup \{\ \sigma_i\sigma_j\sigma \mid\ \sigma_j \in \mathbf{C}[\![P]\!](\sigma_i),\ \sigma_j\sigma \in \mathcal{X}\ \}$$

An *infinite execution trace* $\sigma \in S^\omega[\![P]\!]$ of a program P is an infinite sequence $\sigma_0...\sigma_i... \in \Sigma^\omega$ of length $|\sigma| = \omega$, such that each state σ_{i+1} is a successor of the previous state, i.e., $\sigma_{i+1} \in \mathbf{C}(\sigma_i)$. $S^\omega[\![P]\!]$ can be computed as $gfp^{\subseteq} F^\omega[\![P]\!]$, where $F^\omega[\![P]\!] : \wp(\Sigma^\omega[\![P]\!]) \to \wp(\Sigma^\omega[\![P]\!])$ is defined as:

$$F^\omega[\![P]\!](\mathcal{X}) \stackrel{\text{def}}{=} \{\ \sigma_i\sigma_j\sigma \mid\ \sigma_j \in \mathbf{C}[\![P]\!](\sigma_i),\ \sigma_j\sigma \in \mathcal{X}\ \}$$

As usual, the *maximal trace semantics* $S^\infty[\![P]\!] \in \wp(\Sigma^\infty)$ of program P is given by $S^\infty[\![P]\!] \stackrel{\text{def}}{=} S^+[\![P]\!] \cup S^\omega[\![P]\!]$.

3 Code obfuscation as semantic transformation

Code obfuscation is defined as a *potent* program transformation that preserves the *observational behaviour* of programs [5, 7, 8], where potent means that the transformed (obfuscated) program is harder to understand than the original one. It is clear that the standard definition of code obfuscation relies on the notion of potent transformation, and therefore on a fixed metric for measuring program complexity, which is an old problem [20, 26]. In the literature there are several different metrics for program complexity that can be used according to the current need. For example, the complexity of a program can be measured by the length of the program (the number of instructions and arguments) [26], by the nesting level (the number of nested conditions) [27], or by the data flow (the number of references to local variables) [34]. Given a metric for program complexity it is possible to measure the potency of a transformation, namely how much more difficult is the transformed program to understand than the original one. It is clear that, in order to design a good obfuscator, the potency of the transformation should be maximized.

Definition 3 *[5, 7, 8] A program transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is an obfuscator if:*

1. *the transformation $\mathbb{t}$ is potent and*
2. *P and $\mathbb{t}[\![P]\!]$ have the same observational behaviour, i.e., if P fails to terminate or it terminates with an error condition then $\mathbb{t}[\![P]\!]$ may or may not terminate; otherwise $\mathbb{t}[\![P]\!]$ must terminate and produce the same output as P.*

Point 2 of the above (informal) definition requires the original and obfuscated program to behave equivalently whenever P terminates, whereas no constraints are specified when P diverges. This means that in order to classify a program transformation $\mathbb{t}$ as an obfuscation, we have to analyze the behaviour of the corresponding semantic transformation $t = S \circ \mathbb{t} \circ \mathbb{p}$ only on the finite traces in $S[\![P]\!]$ that terminate with a final/blocking state. Thus, we should focus only on finite traces and consider the maximal finite trace semantics domain Σ^+ instead of $\Sigma^\infty = \Sigma^+ \cup \Sigma^\omega$. Given $\mathcal{X} \subseteq \Sigma^\infty$, we denote with $\mathcal{X}^+$ the set of finite traces of $\mathcal{X}$, i.e., $\mathcal{X}^+ = \mathcal{X} \cap \Sigma^+$, and with $\mathcal{X}^\omega$ the set of infinite traces of $\mathcal{X}$, i.e., $\mathcal{X}^\omega = \mathcal{X} \cap \Sigma^\omega$. Given a transformation $f : \wp(\Sigma^+ \cup \Sigma^\omega) \to \wp(\Sigma^+ \cup \Sigma^\omega)$ we have that, in general, $f(\mathcal{X}^+) \notin \wp(\Sigma^+)$ and $f(\mathcal{X}^\omega) \notin \wp(\Sigma^\omega)$, which means that a transformation on $\wp(\Sigma^+ \cup \Sigma^\omega)$ may not preserve the (non)termination

of the input traces. However, this is not true when speaking of code obfuscation. In fact, point 2 of Definition 3 says that a semantic obfuscator $t = S \circ \mathbb{t} \circ \mathbb{p}$ should transform finite traces into observationally equivalent finite traces, i.e., $\forall \mathcal{X}^+ \in \wp(\Sigma^+) : t(\mathcal{X}^+) \in \wp(\Sigma^+)$ and $\alpha_{\mathcal{O}}(t(\mathcal{X}^+)) = \alpha_{\mathcal{O}}(\mathcal{X}^+)$, where $\alpha_{\mathcal{O}}$ models the observation. In particular, point 2 of Definition 3 is interested in preserving the input-output behaviour of terminating computations, and it can be restated in terms of $t = S \circ \mathbb{t} \circ \mathbb{p}$ as follows:

$$\forall P \in \mathbb{P}, \forall \sigma \in S^+[\![P]\!] : \exists \eta \in t(S^+[\![P]\!]) : \sigma_0 = \eta_0 \wedge \sigma_f = \eta_f$$

It is possible to show that the semantic transformations that correspond to common obfuscating algorithms such as opaque predicate insertion, semantic nop insertion, variable renaming, substitution of equivalent commands and code reordering satisfy the above condition. Thus, when considering the semantic aspects of an obfuscation $\mathbb{t}$ that satisfies Definition 3, we focus only on the effects that the obfuscation has on finite traces and consider the restriction of $t = S \circ \mathbb{t} \circ \mathbb{p}$ to $\wp(\Sigma^+)$, i.e., $t|_{\wp(\Sigma^+)} : \wp(\Sigma^+) \to \wp(\Sigma^+)$. In order to simplify the notation, from now on we will write $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$ instead of $t|_{\wp(\Sigma^+)} : \wp(\Sigma^+) \to \wp(\Sigma^+)$.

Given a set of finite traces, i.e., a maximal finite trace semantics, it is possible to derive the corresponding set of commands, i.e., the corresponding program, by collecting all the commands that occur in the given traces [16]. This is formalized by function $\mathbb{p}^+ : \wp(\Sigma^+) \to \mathbb{P}$ that maps set of traces in set of commands. In particular, $\mathbb{p}^+$ is defined as follows:

$$\mathbb{p}^+[\mathcal{X}] \stackrel{\text{def}}{=} \left\{ C \,\middle|\, \exists \sigma \in \mathcal{X} : \exists i \in [0, |\sigma|[: \exists \rho \in \mathfrak{E} : \sigma_i = \langle \rho, C \rangle \right\}$$

Since we are only interested in the effects of obfuscation on finite traces, from now on we consider the following specification of the Galois insertion (1) that defines the relation between programs and their maximal finite trace semantics:

$$\langle \wp(\Sigma^+), \subseteq \rangle \underset{\mathbb{p}^+}{\overset{S^+}{\leftrightarrows}} \langle \mathbb{P}/_{=}, \subseteq \rangle \tag{2}$$

3.1 Modeling attacks

Code obfuscation aims at preventing malicious host attacks by obstructing the disclosure of sensitive information about proprietary programs. Code obfuscation can provide an important defense against automatic malicious reverse-engineering attacks, but it cannot provide a complete protection against malicious host attacks: a competent programmer, who is willing to invest enough time and effort, will always be able to reverse-engineer any obfuscated program. Thus, in order to understand the limits and potentialities of code obfuscation we need to specify a model for automatic attacks. Automatic reverse-engineering techniques typically consist in static program analysis (e.g., data flow analysis, control flow analysis, alias analysis, program slicing) and dynamic program analysis (e.g., dynamic testing, profiling, program tracing). Static and dynamic program analyses can be formalized as instances of abstract interpretation, which is a general theory for reasoning about program semantics introduced in Section 2.2. Following this observation we model attacks, i.e., static and dynamic program analyzers, as abstract domains $\varphi \in uco(\wp(\Sigma^+))$, where the properties encoded in the abstract domain φ are the ones in which the attacker is interested. In this setting, the complete lattice of abstract domains $\langle uco(\wp(\Sigma^+)), \sqsubseteq \rangle$ provides the right framework where to compare attacks with respect to their degree of abstraction. A coarse

abstraction models an attacker that observes simple semantic properties, while finer abstractions model attackers that are interested in the details of computation. It is clear that what an attacker can deduce from the observation of an obfuscated program depends both on the property of interest of the attacker and on the particular obfuscation used.

In this setting, being able to distinguish the properties, i.e., the abstractions, of program semantics that are not preserved by an obfuscation coincides with the identification of the class of attacks against which the obfuscation is potent. In fact, when an obfuscation $\mathbb{t}$ does not preserve a property $\varphi \in uco(\wp(\Sigma^+))$, i.e., when $\varphi(S^+[\![P]\!]) \neq \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])$, it means that an attacker that analyzes the behaviour of the transformed program $S^+[\![\mathbb{t}[\![P]\!]]\!]$ cannot deduce property φ of the behaviour of the original program $S^+[\![P]\!]$, which means that property φ has been obfuscated by $\mathbb{t}$. If, on the one hand, the fact that $\varphi(S^+[\![P]\!]) \neq \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])$ ensures that obfuscation $\mathbb{t}$ obstructs the disclosure of property φ, namely that $\mathbb{t}$ is potent with respect to φ, on the other hand it does not guarantee that the obfuscation cannot be easily undone, namely that $\mathbb{t}$ is a resilient transformation. In the following, we provide a semantics-based definition of code obfuscation that allows us to characterize the potency of an obfuscation $\mathbb{t}$ in terms of the set of attacks that $\mathbb{t}$ is able to obstruct. The proposed semantics-based definition does not deal with the resilience of obfuscation, namely we do not provide a general framework where to measure how difficult it is for an automatic attack to undo an obfuscation. Thus, while our semantics-based approach to code obfuscation provides a general framework where to compare different transformations with respect to their potency, the resilience of different obfuscations should be analyzed case by case and it might not be possible to compare the resilience of different kind of obfuscations. However, in Section 4.5 we analyze the resilience of control code obfuscation through opaque predicate insertion. In this case, it turns out that the resilience of opaque predicate insertion with respect to a given attack can be measured in terms of completeness of the abstract domain modeling the attack. This result allows us to compare the resilience of the insertion of different opaque predicates with respect to a given attack.

3.2 Semantics-based code obfuscation

If, on the one hand, obfuscating transformations attempt to mask program properties in order to confuse the attackers, on the other hand they must preserve the observational behaviour of programs. According to the standard definition of obfuscation (Definition 3), preservation of the observational behaviour is guaranteed by the preservation of the input-output behaviour of terminating program executions. Recall that program semantics formalizes program behaviour for every possible inputs. The set of all program traces, i.e., the maximal trace semantics, expressing the evolution of program states during every possible computation, is a possible formalization of program behaviour, namely a possible program semantics. In the literature there exist many different program semantics. The most common ones include the big-step, termination and non-termination, Plotkin's natural, Smyth's demonic, Hoare's angelic relational and corresponding denotational, Dijkstra's predicate transformer weakest-precondition and weakest-liberal precondition and Hoare's partial and total axiomatic semantics. In [13] Cousot defines a hierarchy of semantics, where the above semantics are all derived by successive abstractions from the maximal trace semantics. In this framework $uco(\wp(\Sigma^\infty))$ is the lattice of abstract semantics, namely each closure in $uco(\wp(\Sigma^\infty))$ represents an abstraction of the maximal trace semantics. As argued earlier, when dealing with code obfuscation we consider the maximal finite

trace semantics of programs computed on $\wp(\Sigma^+)$, also known as the *angelic* semantics. Observe that the angelic semantics can be formalized as an abstraction of the maximal trace semantics computed on $\wp(\Sigma^\infty)$. In particular, the angelic semantics is obtained by approximating sets of possibly finite and infinite traces with the set of finite traces only, i.e., $\alpha^+ : \wp(\Sigma^\infty) \to \wp(\Sigma^+)$ is defined as $\alpha^+(\mathcal{X}) = \mathcal{X} \cap \Sigma^+$, while $\gamma^+ : \wp(\Sigma^+) \to \wp(\Sigma^\infty)$ is given by $\gamma^+(\mathcal{Y}) = \mathcal{Y} \cup \Sigma^\omega$. Also the *(natural) denotational semantics DenSem*, which abstracts away the history of the computation by observing only the input/output relation of finite traces and the input of diverging computations, can be formalized as an abstract interpretation of the maximal trace semantics: $\mathit{DenSem}(X) = \{\sigma \in \Sigma^+ \mid \exists \delta \in X^+.\ \sigma_0 = \delta_0 \wedge \sigma_f = \delta_f\} \cup \{\sigma \in \Sigma^\omega \mid \exists \delta \in X^\omega . \sigma_0 = \delta_0\}$, where $X^+ \stackrel{\text{def}}{=} X \cap \Sigma^+$ and $X^\omega \stackrel{\text{def}}{=} X \cap \Sigma^\omega$. In this context, the fact that Definition 3 requires the preservation of the input/output denotational semantics on finite traces, i.e., $\mathit{DenSem}(S^+[\![P]\!]) = \mathit{DenSem}(S^+[\![\mathbb{t}[\![P]\!]]\!])$, seems like a restriction on the possible semantic properties that a program transformation could preserve. Our idea is to relax this constraint by providing a definition of code obfuscation which is parametric on the semantic properties to preserve on finite traces.

In order to provide a semantics-based notion of code obfuscation, we need to specify a semantics-based definition of transformation potency.

Definition 4 *A program transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is potent if there is a property $\varphi \in uco(\wp(\Sigma^+))$ and a program $P \in \mathbb{P}$ such that: $\varphi(S^+[\![P]\!]) \neq \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])$.*

The idea is that a program transformation $\mathbb{t}$ is potent when there exists a semantic property $\varphi \in uco(\wp(\Sigma^+))$ that is not preserved by $\mathbb{t}$, namely when there exists a property φ obfuscated by $\mathbb{t}$. In fact, when $\varphi(S^+[\![P]\!]) \neq \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])$ it means that when an attack analyzes the behaviour of the transformed program, it is not able to derive property φ of the behaviour of the original program. We have already observed that when dealing with obfuscations we have that $S^+ \circ \mathbb{t} = t \circ S^+$, which means that the potency of a transformation $\mathbb{t}$ with respect to a property φ can be equivalently expressed in terms of the semantic transformation $t = S^+ \circ \mathbb{t} \circ \mathbb{p}$, i.e., $\mathbb{t}$ is potent with respect to φ when $\varphi(S^+[\![P]\!]) \neq \varphi(t(S^+[\![P]\!]))$.

According to Definition 4 any program transformation that is different from identity would be potent with respect to some property φ. Moreover, given a program transformation $\mathbb{t}$, each semantic property $\varphi \in uco(\wp(\Sigma^+))$ can be classified either as a preserved or as a masked property with respect to $\mathbb{t}$. In order to distinguish between preserved and hidden properties it is useful to define the most concrete property $\delta_{\mathbb{t}} \in uco(\wp(\Sigma^+))$ preserved by a transformation $\mathbb{t}$ on all programs. In order to prove the existence of the most concrete property that a transformation $\mathbb{t}$ preserves, we need to prove that the reduced product between all the abstract domains φ_i, which encodes properties that are preserved by $\mathbb{t}$ on all programs, expresses a property that is preserved by $\mathbb{t}$ on all programs. Thus, given the set $\{\varphi_i\}_{i \in H}$ of the properties preserved by $\mathbb{t}$ on all programs, i.e., $\forall P \in \mathbb{P}, \forall i \in H : \varphi_i(S^+[\![P]\!]) = \varphi_i(S^+[\![\mathbb{t}[\![P]\!]]\!])$, we need to show that $\forall P \in \mathbb{P}$: $(\sqcap_{i \in H} \varphi_i) S^+[\![P]\!] = (\sqcap_{i \in H} \varphi_i) S^+[\![\mathbb{t}[\![P]\!]]\!]$, which is easily proved by the fact that $\forall P \in \mathbb{P}$: $(\sqcap_{i \in H} \varphi_i) S^+[\![P]\!] = \cap_{i \in H} (\varphi_i S^+[\![P]\!]) = \cap_{i \in H} (\varphi_i S^+[\![\mathbb{t}[\![P]\!]]\!]) = (\sqcap_{i \in H} \varphi_i) S^+[\![\mathbb{t}[\![P]\!]]\!]$. Thus, there exists an unique most concrete preserved property and it is computed as the greatest lower bound between the properties preserved by $\mathbb{t}$ on programs:

$$\delta_{\mathbb{t}} \stackrel{\text{def}}{=} \sqcap \{\varphi \in uco(\wp(\Sigma^+)) | \forall P \in \mathbb{P} : \varphi(S^+[\![P]\!]) = \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])\}$$

Equivalently, $\delta_{\mathbb{t}} = \sqcap \{\varphi \in uco(\wp(\Sigma^+)) | \forall P \in \mathbb{P} : \varphi(S^+[\![P]\!]) = \varphi(t(S^+[\![P]\!]))\}$, since we are considering algorithmic transformations. The most concrete preserved property

$\delta_{\mathbb{t}}$ makes it then possible to classify each property $\varphi \in uco(\wp(\Sigma^+))$ either as obfuscated or preserved by transformation $\mathbb{t}$. In particular, every property $\varphi \in uco(\wp(\Sigma^+))$ that is implied by $\delta_{\mathbb{t}}$, i.e., such that $\delta_{\mathbb{t}} \sqsubseteq \varphi$, is preserved by transformation $\mathbb{t}$, while every other property is obfuscated. In particular, $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi)$ precisely expresses what transformation $\mathbb{t}$ obfuscates of property $\varphi \in uco(\wp(\Sigma^+))$. In fact, the least common abstraction $\delta_{\mathbb{t}} \sqcup \varphi$ represents what the two properties have in common; then, by "subtracting" the common part from φ, we obtain what $\mathbb{t}$ hides of the property φ. Specifically, if property φ is preserved, namely if $\delta_{\mathbb{t}} \sqsubseteq \varphi$, then $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) = \top$, while for every obfuscated property we have that $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) \neq \top$, meaning that something about property φ has been lost during transformation $\mathbb{t}$. Following this observation, the set of properties that are masked by a program transformation $\mathbb{t}$ can be formalized as follows:

$$O_{\delta_{\mathbb{t}}} = \{\varphi \in uco(\wp(\Sigma^+)) \mid \varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) \neq \top\}$$

$O_{\delta_{\mathbb{t}}}$ identifies exactly the set of attacks that are obstructed, i.e., defeated, by $\mathbb{t}$. In fact, $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) = \top$ iff $\varphi = \delta_{\mathbb{t}} \sqcup \varphi$ iff $\delta_{\mathbb{t}} \sqsubseteq \varphi$ iff φ is preserved by $\mathbb{t}$. Thus, a program transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ can be seen as an obfuscator that is harmless with respect to any attack modeled by an abstraction φ such that $\delta_{\mathbb{t}} \sqsubseteq \varphi$, and that is powerful with respect to any attack modeled by an abstraction in $O_{\delta_{\mathbb{t}}}$. Hence, the obfuscating behaviour of a transformation $\mathbb{t}$ can be characterized in terms of the most concrete property $\delta_{\mathbb{t}}$ it preserves. This leads us to the following definition of code obfuscation.

Definition 5 *$\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is a δ-obfuscator if $\delta = \delta_{\mathbb{t}}$ and $O_\delta \neq \emptyset$.*

It is worth remarking that this semantics-based definition of code obfuscation is language independent and it models the effects of obfuscation on the trace semantics of any program written in a programming language that can be specified as a transition system. Moreover, when considering attacks as static and dynamic analyzers that are interested in semantic properties of programs, the proposed semantics-based definition of code obfuscation provides a formalization of the informal notion of obfuscator given by Collberg et al. [5, 7] and reported in Definition 3. In fact, every program transformation that is classified as an obfuscator by the standard definition is classified as an obfuscator also by the semantics-based definition. In particular, the class $\mathcal{O}$ of program transformations that are classified as obfuscators following Collberg's definition corresponds to the set of δ-obfuscators where δ is at least the denotational semantics.

Theorem 3 *$\mathcal{O} = \{\delta$-obfuscators $\mid \delta \sqsubseteq$ DenSem$\}$.*

PROOF: We have to show that $\mathcal{O} = \{\mathbb{t} \mid \delta_{\mathbb{t}} \sqsubseteq \textit{DenSem}, O_{\delta_{\mathbb{t}}} \neq \emptyset\}$. The condition $O_{\delta_{\mathbb{t}}} \neq \emptyset$ requires transformation $\mathbb{t}$ to be potent, and it is therefore equivalent to point 1 of Definition 3. Thus, we have to show that the program transformations that preserve at least the *DenSem* of programs are the ones that preserve the observational behaviour as defined in Definition 3, namely that satisfy point 2 of Collberg's definition.

$$\begin{aligned}
&\mathbb{t} : \delta_{\mathbb{t}} \sqsubseteq \textit{DenSem} \\
&\Leftrightarrow \forall P \in \mathbb{P} : \delta_{\mathbb{t}}(S^+[\![P]\!]) = \delta_{\mathbb{t}}(S^+[\![\mathbb{t}[\![P]\!]]\!]) \\
&\Leftrightarrow \forall P \in \mathbb{P}, \forall \sigma \in S^+[\![P]\!], \exists \eta \in S^+[\![\mathbb{t}[\![P]\!]]\!] \text{: } \sigma_0 = \eta_0 \wedge \sigma_f = \eta_f \\
&\Leftrightarrow \mathbb{t} \text{ preserves the observational behaviour according to Definition 3}
\end{aligned}$$

□

The formalization of the notion of code obfuscation introduced by Definition 5 allows us to consider every program transformation as a potential code obfuscator, where the

potency of transformation $\mathbb{t}$ is characterized in terms of the most precise property $\delta_{\mathbb{t}}$ preserved by $\mathbb{t}$. Moreover, it generalizes the standard definition of code obfuscation, where obfuscating transformations are not forced to be *DenSem*-preserving but they can also be more invasive as far as the preserved property maintains enough information with respect to the current need. For example, let us consider an application P that is responsible of keeping updated the total amount tot of the bank account of each client, and an application Q that sends a warning to the bad clients every time their total amount tot corresponds to a negative value. Assume that we are interested in protecting application P through code obfuscation. It is clear that, in order to ensure the proper execution of application Q, the obfuscated version of application P has to preserve (at least) the sign of variable tot . This means that, we can allow obfuscations that loose the observational behaviour of application P but not the sign of variable tot. In this setting, a program transformation that replaces the value of variable tot with its double $2 * tot$ is an obfuscation following our definition, while it is not an obfuscation following Collbergs definition.

Moreover, it is clear that our notion of code obfuscation provides a more precise characterization of the obfuscating behaviour of a program transformation $\mathbb{t}$ even when $\mathbb{t}$ satisfies the Collbergs definition. In fact, while the standard notion of obfuscation only distinguish between transformations that preserve *DenSem* and the ones that do not preserve *DenSem*, our definition of code obfuscation relies on a much finer classification that distinguishes between every possible abstractions of trace semantics.

3.3 Constructive characterization of $\delta_{\mathbb{t}}$

Since the obfuscating behaviour of a program transformation $\mathbb{t}$ is characterized in terms of the most concrete property $\delta_{\mathbb{t}}$ it preserves, it is important to provide a constructive methodology for deriving $\delta_{\mathbb{t}}$ from a given transformation $\mathbb{t}$. We have already observed that, when dealing with algorithmic transformations, a property $\varphi \in uco(\wp(\Sigma^+))$ is preserved by $\mathbb{t}$ if and only if it is preserved by $t = S \circ \mathbb{t} \circ \mathbb{p}$. In this section we refer to the semantic properties preserved by the semantic transformation t since we find it more convenient.

In the following we provide a constructive characterization of the most concrete property preserved by a transformation in terms of its fixpoints. In particular, given a semantic transformation t and a program P, we define a predicate $Pres_{P,t} : \wp(\Sigma^+) \to \{true, false\}$ that identifies the elements of $\wp(\Sigma^+)$ that are fixpoints of a closure operator that encodes a property preserved by t on program P (Lemma 2). Next we show that this property is exactly the most concrete property preserved by t on program P (Theorem 4), and then we show how $\delta_{\mathbb{t}}$ can be obtained as the least upper bound of the most concrete properties preserved by $\mathbb{t} = \mathbb{p}^+ \circ t \circ S^+$ on each program (Theorem 5).

Given a program P and a semantic transformation $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$, we define a domain transformer $K_{P,t} : uco(\wp(\Sigma^+)) \to uco(\wp(\Sigma^+))$ that returns the abstract domain preserved by t on P that is closer to the input domain $\mu \in uco(\wp(\Sigma^+))$.

$$K_{P,t} \stackrel{\text{def}}{=} \lambda\mu.\sqcap\{\varphi \in uco(\wp(\Sigma^+)) \mid \mu \sqsubseteq \varphi \wedge \varphi(S^+[\![P]\!]) = \varphi(t(S^+[\![P]\!]))\}$$

Intuitively $K_{P,t}$ loses the minimal amount of information with respect to a given abstract domain in order to obtain a property preserved by t on P. Consequently, $K_{P,t}(id)$ is the most concrete property preserved by transformation t on program P. By definition $K_{P,t}(id)$ is a closure operator and it is therefore uniquely determined by the set of its fixpoints.

Let us consider the predicate $Pres_{P,t} : \wp(\Sigma^+) \to \{true, false\}$ over set of traces. Given a set of traces $\mathcal{X} \in \wp(\Sigma^+)$ we have that $Pres_{P,t}(\mathcal{X}) = true$ when:

$$S^+[\![P]\!] \subseteq \mathcal{X} \Leftrightarrow t(S^+[\![P]\!]) \subseteq \mathcal{X}$$

Hence, predicate $Pres_{P,t}$ does not distinguish between the set of original traces $S^+[\![P]\!]$ and their obfuscation $t(S^+[\![P]\!])$, namely it does not distinguish between the behaviour of the original and obfuscated program. The following result shows that the elements $\mathcal{X} \in \wp(\Sigma^+)$ that satisfy $Pres_{P,t}$ form an abstract domain that encodes a property that is preserved by transformation t on program P.

Lemma 2 *Given a transformation $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$ and a program $P \in \mathbb{P}$, there exists $\varphi_{P,t} \in uco(\wp(\Sigma^+))$ such that $\varphi_{P,t}(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$, moreover $\varphi_{P,t}$ is preserved by transformation t on program P.*

PROOF: Let us show that $\{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$ is a Moore family. It is clear that $S^+[\![P]\!] \subseteq \Sigma^+ \Leftrightarrow t(S^+[\![P]\!]) \subseteq \Sigma^+$, and therefore $\Sigma^+ \in \{\mathcal{X} \mid Pres_{P,t}(\mathcal{X})\}$ is the top element. We have to show that the set $\{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$ is closed under *glb*, namely that given $\{\mathcal{X}_i\}_{i \in I}$ such that $\forall i \in I : Pres_{P,t}(\mathcal{X}_i) = true$, then $Pres(\cap_{i \in I}\mathcal{X}_i) = true$. In fact we have that $S^+[\![P]\!] \subseteq \cap_{i \in I}\mathcal{X}_i$ iff $\forall i \in I : S^+[\![P]\!] \subseteq \mathcal{X}_i$ iff $\forall i \in I : t(S^+[\![P]\!]) \subseteq \mathcal{X}_i$ iff $t(S^+[\![P]\!]) \subseteq \cap_{i \in I}\mathcal{X}_i$. This proves that there exists a closure operator, denoted $\varphi_{P,t} \in uco(\wp(\Sigma^+))$, such that $\varphi_{P,t}(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$.

Now we have to prove that the property expressed by $\varphi_{P,t} \in uco(\wp(\Sigma^+))$ is preserved by t on P, namely that $\varphi_{P,t}(S^+[\![P]\!]) = \varphi_{P,t}(t(S^+[\![P]\!]))$. Let $\mathcal{X}_1$ be the best approximation of $S^+[\![P]\!]$ in $\varphi_{P,t}(\wp(\Sigma^+))$, namely let $\varphi_{P,t}(S^+[\![P]\!]) = \mathcal{X}_1$. This means that $S^+[\![P]\!] \subseteq \mathcal{X}_1$ and since $Pres_{P,t}(\mathcal{X}_1) = true$ we have that $t(S^+[\![P]\!]) \subseteq \mathcal{X}_1$. Now we have to prove that $\mathcal{X}_1$ is the best approximation of $t(S^+[\![P]\!])$ in $\varphi_{P,t}(\wp(\Sigma^+))$. Assume that $\varphi_{P,t}(t(S^+[\![P]\!])) = \mathcal{X}_2$ where $\mathcal{X}_2 \sqsubset \mathcal{X}_1$, namely that there exists an element $\mathcal{X}_2 \in \varphi_{P,t}(\wp(\Sigma^+))$ that approximates $t(S^+[\![P]\!])$ better than what $\mathcal{X}_1$ does. This means that $t(S^+[\![P]\!]) \subseteq \mathcal{X}_2$ which, since $Pres_{P,t}(\mathcal{X}_2) = true$, implies that $S^+[\![P]\!] \subseteq \mathcal{X}_2$. But this would imply that $\varphi_{P,t}(S^+[\![P]\!]) = \mathcal{X}_2$ which contradicts the hypothesis $\varphi_{P,t}(S^+[\![P]\!]) = \mathcal{X}_1$.

□

The following result shows that the closure operator whose fixpoints are characterized by the predicate $Pres_{P,t}$, i.e., $\varphi_{P,t}(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$, is the most concrete property preserved by transformation t on program P.

Theorem 4 $K_{P,t}(id)(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$.

PROOF: Let us show that $K_{P,t}(id) = \varphi_{P,t}$. By definition $K_{P,t}(id)$ is the most concrete property preserved by t on P, while from Lemma 2, $\varphi_{P,t}$ is a property preserved by t on P, therefore $K_{P,t}(id) \sqsubseteq \varphi_{P,t}$. Thus, we have to show that $\varphi_{P,t} \sqsubseteq K_{P,t}(id)$, namely that $K_{P,t}(id)(\wp(\Sigma^+)) \subseteq \varphi_{P,t}(\wp(\Sigma^+))$. Let us assume that $\exists \mathcal{X} \in K_{P,t}(id)(\wp(\Sigma^+))$ such that $\mathcal{X} \notin \varphi_{P,t}(\wp(\Sigma^+))$. In this case we have that $Pres_{P,t}(\mathcal{X}) = false$, namely that $S^+[\![P]\!] \subseteq \mathcal{X}$ while $t(S^+[\![P]\!]) \not\subseteq \mathcal{X}$, or that $S^+[\![P]\!] \not\subseteq \mathcal{X}$ while $t(S^+[\![P]\!]) \subseteq \mathcal{X}$. Let us consider the case where $S^+[\![P]\!] \subseteq \mathcal{X}$ while $t(S^+[\![P]\!]) \not\subseteq \mathcal{X}$ (the other case is analogous). Let $\mathcal{W}$ be the element of $K_{P,t}(id)(\wp(\Sigma^+))$ that better approximates both $S^+[\![P]\!]$ and $t(S^+[\![P]\!])$, namely let $K_{P,t}(id)(S^+[\![P]\!]) = K_{P,t}(id)(t(S^+[\![P]\!])) = \mathcal{W}$. It is clear that $\mathcal{W} \neq \mathcal{X}$, since we are in the case where $t(S^+[\![P]\!]) \not\subseteq \mathcal{X}$. This implies that

$S^+[\![P]\!] \subseteq \mathcal{W}$ and $t(S^+[\![P]\!]) \subseteq \mathcal{W}$. Since $K_{P,t}(id)(\wp(\Sigma^+))$ is a Moore family we have that $\mathcal{X} \cap \mathcal{W}$ is an element of $K_{P,t}(id)(\wp(\Sigma^+))$. Moreover, it holds that $S^+[\![P]\!] \subseteq \mathcal{X} \cap \mathcal{W}$, while $t(S^+[\![P]\!]) \not\subseteq \mathcal{X} \cap \mathcal{W}$. This would imply that $K_{P,t}(id)(S^+[\![P]\!]) = \mathcal{X} \cap \mathcal{W}$ while $K_{P,t}(id)(t(S^+[\![P]\!])) = \mathcal{W}$, where $\mathcal{X} \cap \mathcal{W} \neq \mathcal{W}$ since $(\mathcal{X} \cap \mathcal{W}) \sqsubset \mathcal{W}$, which leads to the contradiction $K_{P,t}(id)(S^+[\![P]\!]) \neq K_{P,t}(id)(t(S^+[\![P]\!]))$.

□

Therefore, $K_{P,t}(id)(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid Pres_{P,t}(\mathcal{X})\}$ is the most concrete property preserved by the transformation t on program P. Hence, the most concrete property preserved by t on all programs, is given by the least upper bound between the most concrete properties preserved on each program $P \in \mathbb{P}$ by t, i.e., $\bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)$. More precisely the following holds.

Theorem 5 *Let* $\mathbb{t} : \mathbb{P} \to \mathbb{P}$, *then* $\delta_{\mathbb{t}} = \bigsqcup_{P \in \mathbb{P}} K_{P,S^+ \circ \mathbb{t} \circ \mathbb{p}^+}(id)$.

PROOF: Let us first show that $\bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)$ is the most concrete property preserved by transformation t on all programs. (1) $\bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)$ is preserved: observe that given a program $Q \in \mathbb{P}$ then $K_{Q,t}(id) \sqsubseteq \bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)$, by definition $K_{Q,t}(id)$ is preserved by t on Q, therefore $\forall Q \in \mathbb{P} : \bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)(S^+[\![Q]\!]) = \bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)(t(S^+[\![Q]\!]))$. (2) $\bigsqcup_{P \in \mathbb{P}} K_{P,t}(id)$ is the most concrete property preserved by t. Consider $\eta \in uco(\wp(\Sigma^+))$ where $\forall P \in \mathbb{P} : \eta(S^+[\![P]\!]) = \eta(t(S^+[\![P]\!]))$, then $\bigsqcup_{P \in \mathbb{P}} K_{P,t}(id) \sqsubseteq \eta$ iff $\forall P \in \mathbb{P} : K_{P,t}(id) \sqsubseteq \eta$ which is true since $K_{P,t}(id)$ is the most concrete property preserved by t on P. To conclude recall that t is an algorithmic transformation, therefore we can write $t = S^+ \circ \mathbb{t} \circ \mathbb{p}^+$.

□

The proposed characterization of the most concrete property preserved by a program transformation is used in Section 3.5 in order to specify the obfuscating behaviour of constant propagation, and in Section 4.3 in order to formalize the obfuscating behaviour of opaque predicate insertion.

3.4 Comparing transformations

The semantics-based definition of obfuscation allows us to compare obfuscating transformations with respect to their potency, namely according to the most concrete property they preserve. In other words it allows us to formalize a *partial order* relation between obfuscating transformations with respect to the sets of properties hidden by each transformation. On the one hand, it comes natural to think that a transformation $\mathbb{t}$ is more potent than a transformation $\mathbb{t}'$ if it obfuscates more properties, namely if $\mathbb{t}$ defeats more attacks than what $\mathbb{t}'$ does. On the other hand, it may be interesting to know which obfuscation is more potent with respect to a particular attack $\varphi \in uco(\wp(\Sigma^+))$, namely which obfuscation is better to use when we want to obstruct an attack modeled by the abstract domain φ. The idea is that $\mathbb{t}$ is more potent than $\mathbb{t}'$ with respect to φ if $\mathbb{t}$ obfuscates the property φ more than what $\mathbb{t}'$ does, namely if the amount of information that $\mathbb{t}$ looses about property φ is bigger than the one lost by $\mathbb{t}'$.

Definition 6 *Given two transformations* $\mathbb{t}, \mathbb{t}' : \mathbb{P} \to \mathbb{P}$ *and a property* $\varphi \in O_{\delta_{\mathbb{t}}} \cap O_{\delta_{\mathbb{t}'}}$*:*

- $\mathbb{t}$ *is more potent than* $\mathbb{t}'$*, denoted by* $\mathbb{t}' \ll \mathbb{t}$*, if* $O_{\delta_{\mathbb{t}'}} \subseteq O_{\delta_{\mathbb{t}}}$
- $\mathbb{t}$ *is more potent than* $\mathbb{t}'$ *with respect to* φ*, denoted* $\mathbb{t}' \ll_{\varphi} \mathbb{t}$*, if* $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) \sqsubseteq \varphi \ominus (\delta_{\mathbb{t}'} \sqcup \varphi)$

From the structure of the lattice of abstract interpretations $uco(\wp(\Sigma^+))$ it is possible to give an alternative characterization of the set of properties $O_{\delta_{\mathbb{I}}}$ obfuscated by a program transformation. This leads to the observation of some basic properties that relate transformations and preserved properties to the set of masked properties.

Proposition 1 *Let* $\delta, \mu \in uco(\wp(\Sigma^+))$.
- *(1)* $O_\delta = \left\{ \mu \in uco(\wp(\Sigma^+)) \;\middle|\; \mu \notin \uparrow \delta \right\}$
- *(2) If* $\mu \sqsubset \delta$ *then* $O_\mu \subseteq O_\delta$
- *(3)* $O_{\delta \sqcup \mu} = O_\delta \cup O_\mu$

PROOF:
- (1) Recall that given a lattice C and a domain D such that $C \sqsubseteq D$ then $C \ominus D = \top \Leftrightarrow C = D$ [25]. Thus: $\mu \ominus (\delta \sqcup \mu) \neq \top \;\Leftrightarrow\; \delta \sqcup \mu \neq \mu \;\Leftrightarrow\; \mu \notin \uparrow \delta$. Therefore $\{\mu \in uco(\wp(\Sigma^+)) \mid \mu \ominus (\delta \sqcup \mu) \neq \top\}$ is equivalent to $\{\mu \in uco(\wp(\Sigma^+)) \mid \mu \notin \uparrow \delta\}$.
- (2) We have to prove that $\forall \varphi \in O_\mu$ then $\varphi \in O_\delta$. By definition a property φ belongs to O_μ iff $\varphi \in \uparrow \mu = \{\psi \mid \mu \sqsubseteq \psi\}$. By hypothesis $\mu \sqsubset \delta$, therefore if $\delta \sqsubseteq \psi$ then $\mu \sqsubseteq \psi$, therefore $\uparrow \delta \subseteq \uparrow \mu$. This means that if $\varphi \notin \uparrow \mu$ then $\varphi \notin \uparrow \delta$, namely if $\varphi \in O_\mu$ then $\varphi \in O_\delta$.
- (3) We need to show that $\varphi \notin \uparrow \delta \;\wedge\; \varphi \notin \uparrow \mu \;\Leftrightarrow\; \varphi \notin \uparrow (\delta \sqcup \mu)$. This is equivalent to $\varphi \in \uparrow \delta \wedge \varphi \in \uparrow \mu \;\Leftrightarrow\; \varphi \in \uparrow (\delta \sqcup \mu)$, which is true since $\delta \sqsubseteq \varphi \wedge \mu \sqsubseteq \varphi \;\Leftrightarrow\; \delta \sqcup \mu \sqsubseteq \varphi$.

□

In the following section we consider a basic program transformation: the standard constant propagation, and we show how it can be considered as a code obfuscator in the proposed semantics-based framework.

3.5 Case study: Constant propagation

Constant propagation is a well-known program transformation that, knowing the values that are constant at a given program point on all possible executions of a program, propagates these constant values as far forward through the program as possible. The effects of constant propagation on trace semantics have already been studied by Cousot and Cousot in [16], where the authors derive an efficient algorithms for constant propagation as an approximation of the corresponding semantic transformation. In this section we first describe the semantic transformation that performs constant propagation, and then we study its obfuscating behaviour by specifying the most concrete property it preserves.

Semantic aspects of constant propagation [16]. The *residual* $\mathbf{R}[\![D]\!]\rho$ of an arithmetic or boolean expression $D \in \mathbb{E} \cup \mathbb{B}$ in an environment ρ is the expression resulting from specializing D in that environment (see Table 1). When expression D can be fully evaluated in environment ρ, i.e., *var*$[\![D]\!] \subseteq dom(\rho)$, we say that expression D is *static* in the environment ρ, denoted $\mathbf{static}[\![D]\!]\rho$. When D is not static it is *dynamic*. It is clear that $\mathbf{static}[\![D]\!]\rho$ means that the specialization of expression D in environment ρ leads to a static value, i.e., a constant, $\mathbf{R}[\![D]\!]\rho \in \mathfrak{D}_\mathfrak{s} \cup \mathcal{B}_\mathfrak{s}$. Recall that the correctness of expression specialization follows from the fact that given two environments ρ and ρ' such that $dom(\rho) \subseteq dom(\rho')$ and $\forall x \in dom(\rho) : \rho(X) = \rho'(X)$, then $\mathbf{A}[\![\mathbf{R}[\![D]\!]\rho]\!]\rho' = \mathbf{A}[\![D]\!]\rho'$ and in particular $\mathbf{A}[\![\mathbf{R}[\![D]\!]\rho]\!]\rho' = \mathbf{A}[\![\mathbf{R}[\![D]\!]\rho]\!](\rho' \setminus dom(\rho))$. The specialization of action A in environment ρ, denoted as $\mathbf{R}[\![A]\!]\rho$, produces both a residual action and a residual environment as defined in Table 2.

Arithmetic Expressions		$\mathbf{R} \in \mathbb{E} \times \mathfrak{E} \to \mathbb{E}$
$\mathbf{R}[\![n]\!]\rho$	$\stackrel{\text{def}}{=}$	n
$\mathbf{R}[\![X]\!]\rho$	$\stackrel{\text{def}}{=}$	if $X \in dom(\rho)$ then $\rho(X)$ else X
$\mathbf{R}[\![E_1 - E_2]\!]\rho$	$\stackrel{\text{def}}{=}$	let $E_1^r = \mathbf{R}[\![E_1]\!]\rho$ and $E_2^r = \mathbf{R}[\![E_2]\!]\rho$ in if $E_1^r = \mathfrak{z}$ or $E_2^r = \mathfrak{z}$ then $\mathfrak{z}$ else if $E_1^r = n_1$ and $E_2^r = n_2$ then $n = n_1 - n_2$ else $E_1^r - E_2^r$
Boolean Expressions		$\mathbf{R} \in \mathbb{B} \times \mathfrak{E} \to \mathbb{B}$
$\mathbf{R}[\![E_1 < E_2]\!]\rho$	$\stackrel{\text{def}}{=}$	let $E_1^r = \mathbf{R}[\![E_1]\!]\rho$ and $E_2^r = \mathbf{R}[\![E_2]\!]\rho$ in if $E_1^r = \mathfrak{z}$ or $E_2^r = \mathfrak{z}$ then $\mathfrak{z}$ else if $E_1^r = n_1$ and $E_2^r = n_2$ and $b = n_1 < n_2$ then b else $E_1^r < E_2^r$
$\mathbf{R}[\![B_1 \vee B_2]\!]\rho$	$\stackrel{\text{def}}{=}$	let $B_1^r = \mathbf{R}[\![B_1]\!]\rho$ and $B_2^r = \mathbf{R}[\![B_2]\!]\rho$ in if $B_1^r = \mathfrak{z}$ or $B_2^r = \mathfrak{z}$ then $\mathfrak{z}$ else if $B_1^r = \texttt{true}$ or $B_2^r = \texttt{true}$ then $\texttt{true}$ else if $B_1^r = \texttt{false}$ then B_2^r else if $B_2^r = \texttt{false}$ then B_1^r else $B_1^r \vee B_2^r$
$\mathbf{R}[\![\neg B]\!]\rho$	$\stackrel{\text{def}}{=}$	let $B^r = \mathbf{R}[\![B]\!]\rho$ in if $B^r = \mathfrak{z}$ then $\mathfrak{z}$ else if $B^r = \texttt{true}$ then $\texttt{false}$ else if $B^r = \texttt{false}$ then $\texttt{true}$ else $\neg B^r$
$\mathbf{R}[\![\texttt{true}]\!]\rho$	$\stackrel{\text{def}}{=}$	$\texttt{true}$
$\mathbf{R}[\![\texttt{false}]\!]\rho$	$\stackrel{\text{def}}{=}$	$\texttt{false}$

Table 1: Expression Specialization

Let $\alpha^c_{\mathfrak{O}}$ be the *observational abstraction* that has to be preserved by constant propagation in order to ensure the correctness of the transformation. In [16] $\alpha^c_{\mathfrak{O}} : \wp(\Sigma^+) \to \wp(\mathfrak{E}^+)$ is defined as follows:

$$\alpha^c_{\mathfrak{O}}(\mathcal{X}) \stackrel{\text{def}}{=} \{\alpha^c_{\mathfrak{O}}(\sigma) | \sigma \in \mathcal{X}\} \quad \alpha^c_{\mathfrak{O}}(\sigma) \stackrel{\text{def}}{=} \lambda i.\alpha^c_{\mathfrak{O}}(\sigma_i) \quad \alpha^c_{\mathfrak{O}}(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \rho$$

Thus, function $\alpha^c_{\mathfrak{O}}$ abstracts from the particular commands that produce a certain environment evolution keeping only the environment trace. Given a set of finite traces $\mathcal{X} \in \wp(\Sigma^+)$, let $\mathcal{X}^c$ denote the result of a preliminary static analysis detecting constants. Formally $\mathcal{X}^c$ is a sound approximation of $\alpha^c(\mathcal{X})$ where:

$$\alpha^c(\mathcal{X}) = \lambda L.\lambda X. \dot{\bigsqcup} \{\rho(X) \mid \exists \sigma \in \mathcal{X} : \exists C \in \mathbb{C} : \exists i : \sigma_i = \langle \rho, C \rangle, \mathit{lab}[\![C]\!] = L\}$$

where $\dot{\bigsqcup}$ is the pointwise extension of the least upper bound $\bigsqcup$ in the complete lattice $\mathfrak{D}^c \stackrel{\text{def}}{=} \mathfrak{D}_{\mathfrak{z}} \cup \{\top, \bot\}$, where $\forall x \in \mathfrak{D}^c : \bot \sqsubseteq x \sqsubseteq x \sqsubseteq \top$. This means that, given a program P and a label $L \in \mathit{lab}[\![P]\!]$, $\alpha^c(S^+[\![P]\!])(L)$ is an environment mapping (denoted ρ^c_L for short when the set of traces is clear from the context) which, given a variable $X \in \mathit{var}[\![P]\!]$, returns the value of X if X is constant at program point L, $\top$ otherwise. Thus, a variable X of program P has a constant value at program point L when $\alpha^c(S^+[\![P]\!])(L)(X) \neq \top$, i.e., $\rho^c_L(X) \neq \top$.

Actions		$\mathbf{R} \in \mathbb{A} \times \mathfrak{E} \to \mathfrak{E} \times \mathbb{A}$
$\mathbf{R}[\![B]\!]\rho$	$\stackrel{\text{def}}{=}$	$\langle \rho, \mathbf{R}[\![B]\!]\rho \rangle$
$\mathbf{R}[\![X :=?]\!]\rho$	$\stackrel{\text{def}}{=}$	$\langle \rho \setminus X, X :=? \rangle$
$\mathbf{R}[\![X := E]\!]\rho$	$\stackrel{\text{def}}{=}$	if **static**$[\![E]\!]\rho$ then $\langle \rho[X := \mathbf{R}[\![E]\!]\rho], \texttt{skip} \rangle$
		else $\langle \rho \setminus X, X := \mathbf{R}[\![E]\!]\rho \rangle$

Table 2: Action Specialization

The semantic transformation $t^c : \wp(\Sigma^+) \times \alpha^c(\wp(\Sigma^+)) \to \wp(\Sigma^+)$ performing constant propagation is constructively defined as follows:

$$t^c[\mathcal{X}, \mathcal{X}^c] \stackrel{\text{def}}{=} \{t^c[\sigma, \mathcal{X}^c] \mid \sigma \in \mathcal{X}\}$$
$$t^c[\sigma, \mathcal{X}^c] \stackrel{\text{def}}{=} \lambda i.t^c[\sigma_i, \mathcal{X}^c] \quad t^c[\langle \rho, C \rangle, \mathcal{X}^c(\mathit{lab}[\![C]\!])] \stackrel{\text{def}}{=} \langle \rho, t^c[C, \rho^c_{\mathit{lab}[\![C]\!]}] \rangle$$

where command specialization is defined as:

$$t^c[L : A \to L', \rho^c_L] \stackrel{\text{def}}{=} L : t^c[A, \rho^c_L] \to L'$$
$$t^c[A, \rho^c_L] = \text{ let } \langle \rho_r, A_r \rangle \stackrel{\text{def}}{=} \mathbf{R}[\![A]\!]\rho|_{\{X \in \mathbb{X} | \rho^c_L(X) \in \mathfrak{D}_s\}} \text{ in } A_r$$

The correctness of t^c follows from the fact that the transformed traces are valid traces, i.e., $\sigma \in \Sigma^+ \Rightarrow t^c[\sigma, \mathcal{X}^c] \in \Sigma^+$, and that $\alpha^c_{\mathfrak{O}}$ is preserved by t^c since the transformation leaves the environments unchanged [16].

Following the steps elucidated at the end of Section 2.3 it is possible to derive a constant propagation algorithm $\mathbb{t}^c = \mathbb{p}^+ \circ t^c \circ S^+$. We omit here such details because they are not significant for our reasoning.

Obfuscating behaviour of constant propagation. In order to understand the obfuscating behaviour of constant propagation, we need to consider the most concrete property δ_{t^c} preserved by the transformation t^c defined above. Following the characterization proposed by Theorem 5 we can formalize δ_{t^c} as follows:

$$\delta_{t^c} = \bigsqcup_{P \in \mathbb{P}} \{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t^c}(\mathcal{X})\}$$

where, given a set of traces $\mathcal{X} \in \wp(\Sigma^+)$ we have that $\mathit{Pres}_{P,t^c}(\mathcal{X}) = \mathit{true}$ when:

$$S^+[\![P]\!] \subseteq \mathcal{X} \Leftrightarrow \forall S^c[\![P]\!] \sqsupseteq \alpha^c(S^+[\![P]\!]) : t^c[S^+[\![P]\!], S^c[\![P]\!]] \subseteq \mathcal{X}$$

This means that an element $\mathcal{X}$ is a fixpoint of δ_{t^c} if it is not able to distinguish between the semantics of a program and its specialization through constant propagation, whenever constant propagation is performed based on a sound constant analysis. Thus, by specializing with respect to any sound constant analysis every program trace semantics given by a set $\mathcal{Z} \in \wp(\Sigma^+)$, we obtain the set of fixpoints of δ_{t^c}. As shown in Section 3.2, the characterization of the most concrete property δ_{t^c} preserved by constant propagation allows us to classify each attack in $\mathit{uco}(\wp(\Sigma^+))$ either as an harmful or as a succeeding attack.

On the one hand, let us consider the closure operator $\varphi^c_{\mathfrak{O}} = \gamma^c_{\mathfrak{O}} \circ \alpha^c_{\mathfrak{O}}$ corresponding to the observational abstraction $\alpha^c_{\mathfrak{O}}$, where $\gamma^c_{\mathfrak{O}}$ is the concretization map induced by abstraction $\alpha^c_{\mathfrak{O}}$. It is clear that, since the observational abstraction $\alpha^c_{\mathfrak{O}}$ is preserved by

```
a:= 1; b:=2; c:=3; d:=3; e:=0;
while B do
    b:=2*a; d:=d+1; e:=e-a;
    a:=b-a; c:=e+d;
endw
```

L_1 : `a:= 1; b:=2; c:=3; d:=3; e:=0;` $\rightarrow L_2$
L_2 : `B` $\rightarrow L_3$
L_2 : $\neg$`B` $\rightarrow L_5$
L_3 : `b:=2*a; d:=d+1; e:=e-a;` $\rightarrow L_4$
L_4 : `a:=b-a; c:=e+d;` $\rightarrow L_2$
L_5 : `stop` $\rightarrow \not/$

Table 3: A simple program from [11]

t^c, then $\varphi^c_{\mathfrak{O}} \in uco(\wp(\Sigma^+))$ is preserved by transformation t^c. Thus, by definition of δ_{t^c}, we have that $\delta_{t^c} \sqsubseteq \varphi^c_{\mathfrak{O}}$ and therefore $\varphi^c_{\mathfrak{O}} \ominus (\varphi^c_{\mathfrak{O}} \sqcup \delta_{t^c}) = \top$, which, from a code obfuscation point of view, means that property $\varphi^c_{\mathfrak{O}}$ is not obfuscated by constant propagation. In fact, property $\varphi^c_{\mathfrak{O}}$ of the original program traces can be precisely learned also by the analysis of the obfuscated traces, i.e., from $t^c[S^+[\![P]\!], S^c[\![P]\!]]$ for any $S^c[\![P]\!]$ that correctly approximated $\alpha^c(S^+[\![P]\!])$.

On the other hand, every property $\varphi \in uco(\wp(\Sigma^+))$ such that $\varphi \ominus (\varphi \sqcup \delta_{t^c}) \neq \top$ is not preserved by constant propagation, meaning that the attacks modeled by these abstractions are obstructed by constant propagation. In fact, whenever φ is such that $\varphi \ominus (\varphi \sqcup \delta_{t^c}) \neq \top$, it means that one of the following holds:

- $\exists \mathcal{X} \in \varphi(\wp(\Sigma^+)), \exists P \in \mathbb{P}, \exists S^c[\![P]\!] : \alpha^c(S^+[\![P]\!]) \sqsubseteq S^c[\![P]\!]$ such that $S^+[\![P]\!] \subseteq \mathcal{X}$ while $t^c[S^+[\![P]\!], S^c[\![P]\!]] \not\subseteq \mathcal{X}$
- $\exists \mathcal{X} \in \varphi(\wp(\Sigma^+)), \exists P \in \mathbb{P}, \exists S^c[\![P]\!] : \alpha^c(S^+[\![P]\!]) \sqsubseteq S^c[\![P]\!]$ such that $S^+[\![P]\!] \not\subseteq \mathcal{X}$ while $t^c[S^+[\![P]\!], S^c[\![P]\!]] \subseteq \mathcal{X}$

In both cases we have that $\varphi(S^+[\![P]\!]) \neq \varphi(t(S^+[\![P]\!]))$, which means that property φ is able to distinguish the trace semantics of the original program from the trace semantics of the obfuscated one, namely that property φ of the behaviour of the original program cannot be derived by the analysis of the behaviour of the obfuscated program. Hence, constant propagation is a potent transformation with respect to an attack modeled by φ.

As an example of an attack obstructed by constant propagation, let us consider property $\theta \in uco(\wp(\Sigma^+))$, observing the environments and the type of the actions, namely:

$$\alpha_\theta(\mathcal{X}) \stackrel{\text{def}}{=} \left\{ \alpha_\theta(\sigma) \,\middle|\, \sigma \in \mathcal{X} \right\} \qquad \alpha_\theta(\sigma) \stackrel{\text{def}}{=} \lambda i.\alpha_\theta(\sigma_i)$$
$$\alpha_\theta(\langle \rho, C \rangle) \stackrel{\text{def}}{=} (\rho, \mathit{type}[\![\mathit{act}[\![C]\!]]\!])$$

where type maps actions into the following set of action types $\{\mathit{assign}, \mathit{skip}, \mathit{test}\}$. It is clear that this property is not preserved by t^c, since in general $\mathit{type}[\![A]\!] \neq \mathit{type}[\![\mathbf{R}[\![A]\!]\rho]\!]$ (see Example 1). This means that property θ is obfuscated by constant propagation, namely $\theta \in O_{\delta_{t^c}}$, i.e., $\theta \ominus (\theta \sqcup \delta_{t^c}) \neq \top$. This means that $O_{\delta_{t^c}} \neq \emptyset$, thus t^c is an δ_{t^c}-obfuscator following Definition 5.

Example 1 *As observed above, property θ is not preserved by t^c, namely it could happen that: $\theta(S^+[\![P]\!]) \neq \theta(t^c[S^+[\![P]\!], S^c[\![P]\!]])$. In the following we represent the environment as a tuple $(v_a, v_b, v_c, v_d, v_e)$ of values corresponding to the variables a, b, c, d, e in a certain execution point. Let us run the program in Table* 3*, and consider the states $\sigma_2 = \langle(1,2,3,3,0), L_3 : b := 2 * a; d := d + 1;\ e := e - a \rightarrow L_4\rangle$ and $\sigma_3 = \langle(1,2,3,4,-1), L_4 : a := b - a; c := e + d \rightarrow L_2\rangle$. Their transformed versions are: $t^c(\sigma_2) = \langle(1,2,3,3,0), L_3 : d := d + 1;\ e := e - a \rightarrow L_4\rangle$ and $t^c(\sigma_3) = \langle(1,2,3,4,-1), L_4 :$ `skip` $\rightarrow L_2\rangle$. In this case $\theta(\sigma_2) = \langle(1,2,3,3,0), L_3, L_4, assign\rangle$ and $\theta(\sigma_3) = \langle(1,2,3,4,-1), L_4, L_2, assign\rangle$; while, considering the transformed states, $\theta(t^c(\sigma_2)) = \langle(1,2,3,3,0), L_3,\ L_4, assign\rangle$ and $\theta(t^c(\sigma_3)) = \langle(1,2,3,4,-1), L_4, L_2,\ skip\rangle$, showing that the property θ is not preserved.*

Moreover, we can show that what transformation t^c hides of property θ is the type of actions. In fact, consider the closure $\eta \in uco(\wp(\Sigma^+))$ which observes the $type$ of actions:

$$\eta = \lambda\mathcal{X}.\ \{\sigma \mid \sigma' \in \mathcal{X} \text{ and } \forall i.\ \sigma_i = \langle\rho_i, C_i\rangle, \sigma'_i = \langle\rho'_i, C'_i\rangle : type(C_i) = type(C'_i)\}$$

Theorem 6 $\theta \ominus (\theta \sqcup \delta_{t^c}) = \eta$.

PROOF: Let us prove that $\theta \sqcup \delta_{t^c} = \varphi^c_{\mathcal{O}}$. By definition of δ_{t^c} it follows that $\delta_{t^c} \sqsubseteq \varphi^c_{\mathcal{O}}$. Let us show that $\theta \sqsubseteq \varphi^c_{\mathcal{O}}$, namely that $\theta(\wp(\Sigma^+)) \subseteq \varphi^c_{\mathcal{O}}(\wp(\Sigma^+))$.

$$\theta(\mathcal{X}) = \{\sigma \mid \sigma' \in \mathcal{X} \text{ and } \forall i.\ \sigma_i = \langle\rho_i, C_i\rangle, \sigma'_i = \langle\rho_i, C'_i\rangle : type(C_i) = type(C'_i)\}$$

$$\varphi^c_{\mathcal{O}}(\mathcal{X}) = \{\sigma \mid \sigma' \in \mathcal{X} \text{ and } \forall i.\ \sigma_i = \langle\rho_i, C_i\rangle, \sigma'_i = \langle\rho_i, C'_i\rangle\}$$

Thus $\forall\mathcal{X} \in \wp(\Sigma^+) : \theta(\mathcal{X}) \subseteq \varphi^c(\mathcal{X})$ and therefore $\theta \sqsubseteq \varphi^c_{\mathcal{O}}$. Moreover $\varphi^c_{\mathcal{O}}$ is the most concrete property that θ and δ_{t^c} have in common. In fact it is clear that $\theta = \varphi^c_{\mathcal{O}} \sqcap \eta$, and since the $type$ of actions, i.e., η, is not preserved by t^c we have that θ and δ_{t^c} share only the observation of the environments. Hence, we have that $\theta \ominus (\theta \sqcup \delta_{t^c}) = \theta \ominus \varphi^c_{\mathcal{O}} = (\varphi^c_{\mathcal{O}} \sqcap \eta) \ominus \varphi^c_{\mathcal{O}} = \eta$. Where the last equation holds since η is the most abstract domain which reduced product with $\varphi^c_{\mathcal{O}}$ returns θ.

□

4 Control code obfuscation

By *control code obfuscators* we refer to obfuscating techniques that act by masking the control flow behaviour of the original program. These transformations are often based on the insertion of opaque predicates. Two major types of opaque predicates exist: true opaque predicates P^T that always evaluate to $true$, and false opaque predicates P^F that always evaluate to $false$. Figure 3 shows these types of opaque predicates, where solid lines indicate paths that are always taken and dashed lines paths that will never be taken. 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. Consider, for example, the insertion of a branch instruction controlled by an opaque predicate P^T. In this case the true path starts with the next action of the original program, while the false path leads to termination or to buggy code. This confuses the attack that is not aware of the always-true

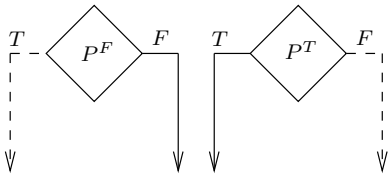

Figure 3: Opaque predicates

evaluation of the opaque predicate, and it has to consider both paths as possible. It is clear that this transformation does not affect program semantics, since at run time the opaque predicate is always evaluated to $true$ and therefore the true path is the only one to be executed. Opaque predicate insertion aims at confusing the program control flow, which may not have significant effects on program trace semantics (since control flow is an abstraction of trace semantics). The insertion of true and false opaque predicates might be detected by an attack that monitors the execution of the program and observes that a certain predicate always evaluates to *true* or to *false*. In order to overcome this limitation Palsberg et al. [36] introduced the notion of *correlated opaque predicates* as a possible improvement over the standard opaque predicates presented above. The idea is to define a family of correlated predicates which evaluate to the same value in any single program run, but this value might vary over different program runs. It is clear that the opaqueness of correlated opaque predicates is difficult to disclose even for an attack that monitors program execution. The notion of dynamic opaque predicate has then been extended to *temporary unstable* or *distributed* opaque predicates in a distributed environment [32]. The value of a temporary unstable opaque predicate may change in different program points during the same run of the program. The idea is that the opaque predicate value depends on predetermined embedded message communication patterns between different processes that maintain the opaque predicate.

In this section we focus on the semantic aspects of the insertion of standard opaque predicates, namely on the insertion of opaque predicates that evaluates to *true* or to *false* during every execution of the program. Once we have understand the limits and potentiality of standard opaque predicates we could discuss how it might be possible to extend our reasoning to correlated opaque predicates and temporary unstable opaque predicates. In particular, in the rest of this section we consider opaque predicates that always evaluate to *true*. It is clear that every result that we obtain in the case of true opaque predicates can be restated in an analogous way for false opaque predicates. Thus, from now on the term opaque predicate will refer to a standard opaque predicate that always evaluates to *true*.

In the following, we start by defining the semantic transformation t^{OP} that mimics the effects of opaque predicate insertion on program trace semantics. In particular, t^{OP} transforms the maximal finite trace semantics of the original program by simply adding opaque tests that always evaluate to *true*, and this clearly modifies the structure of traces. Following the methodology proposed by Cousot and Cousot in [16], and elucidated in Section 2.3, we derive from the semantic transformation t^{OP} its syntactic counterpart $\mathbb{p}^+ \circ t^{OP} \circ S^+$. Next, we extend $\mathbb{p}^+ \circ t^{OP} \circ S^+$ in order to obtain an algorithm $\mathbb{t}^{OP} : \mathbb{P} \to \mathbb{P}$ that performs the insertion of true opaque predicates. In particular, the syntactic transformation $\mathbb{t}^{OP}$ inserts true opaque predicates (as $\mathbb{p}^+ \circ t^{OP} \circ S^+$) together with their potential false path (added manually to $\mathbb{p}^+ \circ t^{OP} \circ S^+$). Given the semantic understanding of true opaque predicate insertion, we study in details the obfuscating

behaviour of this transformation with respect to attacks modeled, as usual, by abstract domains.

4.1 Semantic opaque predicate insertion

Let $\mathfrak{I} : \mathbb{P} \to \wp(\mathbb{L})$ be the result of a preliminary static analysis that given a program returns the subset of its labels, i.e., program points, where it is possible to insert opaque predicates. Usually the preliminary static analysis consists of a combination of liveness analysis and static analyses. On the one hand, liveness analysis is typically used to ensure that no dependencies are broken by the inserted predicate and that the obfuscated program is functionally equivalent to the original one. On the other hand, static analyses such as constant propagation may be used to check whether opaque predicates have definitive values *true* (or *false*), namely if the predicate can be trivially broken. Given a program P, we assume to know the set $\mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!]$ of labels where we are allowed to insert opaque predicates.

Let OP be a set of true opaque predicates. We define the semantic transformation $t^{OP} : \wp(\Sigma^+) \times \wp(\mathbb{L}) \to \wp(\Sigma^+)$ that inserts true opaque predicates $P^T \in OP$ in the traces in $\mathcal{X} \in \wp(\Sigma^+)$ at program points identified by the allowed locations in $\mathbb{K} \in \wp(\mathbb{L})$. In particular, the semantic transformation that performs the insertion of true opaque predicates P^T from the set OP is defined as follows:

$$t^{OP}[\mathcal{X}, \mathbb{K}] \stackrel{\text{def}}{=} \{t^{OP}[\sigma, \mathbb{K}] \mid \sigma \in \mathcal{X}\}$$

$t^{OP}[\langle \rho, L : A \to L' \rangle \sigma, \mathbb{K}] \stackrel{\text{def}}{=}$

$$\begin{cases} \langle \rho, L : A \to L' \rangle \; t^{OP}[\sigma, \mathbb{K}] & \text{if } L \notin \mathbb{K} \\ \langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \; t^{OP}[\sigma, \mathbb{K}] & \text{if } L \in \mathbb{K} \end{cases}$$

Here $\tilde{L}$ denotes an unused location: one that is not present in any of the commands that occur in the traces in $\mathcal{X}$, i.e., $\tilde{L} \notin lab[\![\mathbb{p}^+(\mathcal{X})]\!]$. By definition, transformation t^{OP} changes each trace of $\mathcal{X}$ independently and state by state. In particular, let L be a candidate label for opaque predicate insertion, and let $\langle \rho, L : A \to L' \rangle$ be the (original) program state whose command is labeled by L. Transformation t^{OP} inserts the true opaque predicate P^T at the candidate label L with co-label $\tilde{L}$, which results in the transformed state $\langle \rho, L : P^T \to \tilde{L} \rangle$. In order to preserve program functionality, action A has to be the first action of the true branch of the opaque predicate P^T. This is guaranteed by inserting the new state $\langle \rho, \tilde{L} : A \to L' \rangle$. Thus, transformation t^{OP} performs the insertion of a true opaque predicate P^T by replacing state $\langle \rho, L : A \to L' \rangle$ with the two states $\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle$. It is clear that program environment remains unchanged since test actions, such as opaque predicates, do not affect the value of variables (at least in our model). Figure 4 shows how program traces are modified by opaque predicate insertion: the white dots denote the states of the original trace, while the black dots denote the states corresponding to the inserted opaque tests.

It is clear that the semantic transformation t^{OP}, that transforms traces by inserting true opaque predicates from OP in the allowed program points ($\in \mathbb{K}$), transforms finite traces into finite traces (as shown by the following result).

Lemma 3 *Given $\sigma \in \Sigma^+$ and $\mathbb{K} \in \wp(\mathbb{L})$, then $t^{OP}[\sigma, \mathbb{K}] \in \Sigma^+$.*

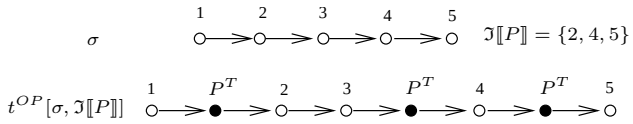

Figure 4: Semantic opaque predicate insertion

PROOF: Given $\sigma \in \Sigma^+$, let $|\sigma| = n$. Observe that $\forall i \in [1, n-2]$ the transformation of the subtrace $\sigma_{i-1}\sigma_i\sigma_{i+1}$ of σ is still a trace, i.e., $t^{OP}[\sigma_{i-1}\sigma_i\sigma_{i+1}, \mathbb{K}] = \sigma'_{i-1}t^{OP}[\sigma_i\sigma_{i+1}, \mathbb{K}] \in \Sigma^+$. Two are the cases that we have to consider. (1) If $L_i \notin \mathbb{K}$, then we have that $\sigma'_{i-1}t^{OP}[\sigma_i\sigma_{i+1}, \mathbb{K}] = \sigma'_{i-1}\sigma_i\sigma'_{i+1}$, and $\sigma_i \in \mathbf{C}(\sigma'_{i-1})$, $\sigma'_{i+1} \in \mathbf{C}(\sigma_i)$ follow form $\sigma \in \Sigma^+$; (2) on the other hand if $L_i \in \mathbb{K}$, we have that:

$$\begin{aligned} \sigma'_{i-1}t^{OP}[\sigma_i\sigma_{i+1}, \mathbb{K}] &= \sigma'_{i-1}\langle \rho_i, L_i : P^T \to \tilde{L}_i\rangle\langle \rho_i, \tilde{L}_i : A_i \to L_{i+1}\rangle\sigma'_{i+1} \\ &= \sigma'_{i-1}\sigma^a_i\sigma^b_i\sigma'_{i+1} \end{aligned}$$

where $\sigma^a_i = \langle \rho_i, L_i : P^T \to \tilde{L}_i\rangle$ and $\sigma^b_i = \langle \rho_i, \tilde{L}_i : A_i \to L_{i+1}\rangle$. The test action given by the opaque predicate does not change the state environment and it is clear that $\sigma^a_i \in \mathbf{C}(\sigma'_{i-1})$, $\sigma^b_i \in \mathbf{C}(\sigma^a_i)$ and $\sigma'_{i+1} \in \mathbf{C}(\sigma^b_i)$. This holds also for the initial and final state, in fact if $L_0 \in \mathbb{K}$ then $\sigma_1 \in \mathbf{C}(\sigma^b_0)$ and if $L_{n-1} \in \mathbb{K}$ then $\sigma^b_{n-1} \in \mathbf{C}(\sigma_{n-2})$. This proves that given $\eta = t^{OP}[\sigma, \mathbb{K}]$ then $\forall i$: $\eta_i \in \mathbf{C}(\eta_{i-1})$. Moreover if $|\mathbb{K}| = h$ then $|\eta| = n + h = k$, thus $\eta \in \Sigma^+$.

□

4.2 Syntactic opaque predicate insertion

Given the semantic transformation t^{OP} it is possible, following the procedure elucidated in Section 2.3, to derive the syntactic transformation performing the insertion of true opaque predicates form the set OP. In particular, transformation $\mathbb{p}^+ \circ t^{OP} \circ S^+$ simply inserts in a program commands which actions are true predicates from OP. Such syntactic transformation can be easily extended to perform code obfuscation based on opaque predicates insertion (denoted as $\mathbb{t}^{OP}$ in the following), by inserting in the transformed program also the dead code forming the false branch of P^T. In fact, following the definition of $\mathbb{p}^+$, these instructions cannot be present in $\mathbb{p}^+ \circ t^{OP} \circ S^+$, since the commands of the never-executed false path are not present in the transformed program semantics.

In the following we describe an opaque predicate insertion algorithm obtained by extending (with the insertion of false paths) the algorithm systematically derived through the methodology explained in Section 2.3 (details of the derivation are in the Appendix). Let us denote with B a set of commands composing a possible false path of a true opaque predicate (never executed at run time), and with *lab*$[\![B]\!]$ the label of the starting point of the execution of B. Let B range over a given collection of programs $\mathfrak{B} \subseteq \wp(\mathbb{C})$, and let $New \subseteq \mathbb{L}$ be a set of "new" program labels.

Opaque$(P, \mathfrak{I}[\![P]\!], New, OP, \mathfrak{B})$
$Q = \emptyset$
$T = \{\ C \in P \mid \mathit{suc}[\![C]\!] \in \mathfrak{L}[\![P]\!]\ \}$
while there exists an unmarked command $L : A \to L'$ in T do
  mark $L : A \to L'$
  if $L \in \mathfrak{I}[\![P]\!]$
    then take $\tilde{L} \in New$
      $New = New \setminus \tilde{L}$
      let $P^T \in OP$
$(*)$ let $B \in \mathfrak{B}$
      $Q = Q \cup \{L : P^T \to \tilde{L}; \tilde{L} : A \to L'\}$
$(*)$ $Q = Q \cup \{L : \neg P^T \to \mathit{lab}[\![B]\!]\}$
    else $Q = Q \cup \{L : A \to L'\}$
  $T = T \cup \{\ C \in P \mid \exists C' \in T : \mathit{suc}[\![C]\!] = \mathit{lab}[\![C']\!]\ \}$

The algorithm **Opaque** considers each command $L : A \to L'$ of the original program P, if L is a candidate label for opaque predicate insertion, i.e., if $L \in \mathfrak{I}[\![P]\!]$, the commands $L : P^T \to \tilde{L}$, $\tilde{L} : A \to L'$ and $L : \neg P^T \to \mathit{lab}[\![B]\!]$, encoding opaque predicate insertion are added to the set Q (initially empty), otherwise the original command $L : A \to L'$ is added to Q. In particular, command $L : \neg P^T \to \mathit{lab}[\![B]\!]$ encodes the false branch of the true opaque predicate and inserts a fake branch connecting the control flow of the original program to the control flow of the never executed code starting at label $\mathit{lab}[\![B]\!]$. In the end, the set Q corresponds to the obfuscated program. It is clear that $|New| \geq |\mathfrak{I}[\![P]\!]|$. Observe that the lines denoted by $(*)$, encoding the insertion of commands forming the false path of the true opaque predicate, have been added manually to $\mathbb{p}^+ \circ t^{OP} \circ S^+$. This happens because the false path of a true opaque predicate is never executed and therefore its commands are not present in the transformed program semantics. In fact, the insertion of an opaque predicate inserts "dead code" in the program (i.e., code that is never executed) and, by definition, the abstraction $\mathbb{p}^+$ cannot return such dead code. Let us denote with $\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$ the extended syntactic transformation corresponding to algorithm **Opaque** reported above. Observe that, if on the one hand $\mathbb{p}^+(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]) = \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$ since they have the same trace semantics, on the other hand $\mathbb{p}^+(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]) \subseteq \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$, since the term on the right contains also the commands of the false paths of the inserted true opaque predicates.

4.3 Obfuscating behaviour of opaque predicate insertion

In order to study the obfuscating behaviour of the insertion of true opaque predicates we need to define the most concrete property preserved by t^{OP}. Following Theorem 5 we have that the most concrete property preserved by opaque predicate insertion can be characterized as follows:

$$\delta_{t^{OP}} = \bigsqcup_{P \in \mathbb{P}} \{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P, t^{OP}}(\mathcal{X})\}$$

where, given $\mathcal{X} \in \wp(\Sigma^+)$, we have that $\mathit{Pres}_{P, t^{OP}}(\mathcal{X}) = \mathit{true}$ when:

$$S^+[\![P]\!] \subseteq \mathcal{X} \ \Leftrightarrow\ \forall \mathfrak{I}[\![P]\!] \subseteq \mathit{lab}[\![P]\!] : t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \mathcal{X}$$

This means that a set of traces $\mathcal{X}$ is "preserved" by opaque predicate insertion if $\mathcal{X}$ contains all the traces that can be obtained from the opaque-free traces in $\mathcal{X}$ by inserting opaque predicates from OP at program points indicated by any preliminary static

analysis, and if for every trace in $\mathcal{X}$ that contains opaque predicates from OP then also the corresponding opaque-free trace belongs to $\mathcal{X}$. As expected, the attack that observes the concrete semantics of program behaviour is confused by opaque predicate insertion, since $S^+[\![P]\!] \neq S^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$, while the attack that observes the denotational semantics of programs is insensible to opaque predicate insertion, since $\delta_{t^{OP}} \sqsubseteq DenSem$ and $DenSem(S^+[\![P]\!]) = DenSem(S^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$.

As noticed above we have that, in general, $S^+[\![P]\!] \neq S^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$, namely that $S^+[\![P]\!] \neq t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]$. In fact, the transformed semantics contains all the traces of the original semantics with some extra states denoting opaque predicate execution as described by the black dots in Figure 4. It is clear that there is no significant information hidden by this obfuscation to attacks that know the concrete program semantics. In fact, by the observation of the concrete semantics, an attack can easily derive the set of inserted opaque predicates and deobfuscate the program. In fact, by knowing the set OP of inserted opaque predicates, we can easily define the trace transformation $d_{OP} : \wp(\Sigma^+) \to \wp(\Sigma^+)$ that recovers the original program trace semantics from the obfuscated one.

$$d_{OP}(\mathcal{X}) \stackrel{\text{def}}{=} \{d_{OP}(\sigma) \mid \sigma \in \mathcal{X}\} \quad d_{OP}(\sigma) \stackrel{\text{def}}{=} \epsilon\, d_{OP}(\sigma)$$

$d_{OP}(\langle \rho, C\rangle\langle \rho', C'\rangle\eta) \stackrel{\text{def}}{=}$

$$\begin{cases} \langle \rho, C\rangle\, d_{OP}(\langle \rho', C'\rangle\eta) & \text{if } \mathit{act}[\![C]\!] \notin OP \\ d_{OP}(\langle \rho, \mathit{lab}[\![C]\!] : \mathit{act}[\![C']\!] \to \mathit{suc}[\![C']\!]\rangle\eta) & \text{if } \mathit{act}[\![C]\!] \in OP \end{cases}$$

It is not surprising that transformation d_{OP}, given the set of inserted opaque predicates, is able to restore the original program semantics. The following result shows that transformation d_{OP} acts as a deobfuscator with respect to the insertion of true opaque predicates from the set OP.

Theorem 7 $S^+[\![P]\!] = d_{OP}(S^+[\![P]\!]) = d_{OP}(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]])$.

PROOF: Let us assume, as usual, that program P has not been previously obfuscated by opaque predicate insertion. By definition of d_{OP} we have that $d_{OP}(S^+[\![P]\!]) = S^+[\![P]\!]$, since $\forall \sigma \in S^+[\![P]\!], \forall \sigma_i = \langle \rho_i, C_i\rangle \in \sigma : \mathit{act}[\![C_i]\!] \notin OP$. On the other hand, we have that $d_{OP}(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]) = \{d_{OP}(\eta) \mid \eta \in t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]\}$. By definition, given $\eta \in t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]$, there exists $\sigma \in S^+[\![P]\!]$ such that $\eta = t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]$. In order to conclude the proof we show that $d_{OP}(\eta) = \sigma$, namely that $d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) = \sigma$. In general $\sigma = \mu^1\sigma_i\mu^2\sigma_j\mu^3...\mu^l$, where $\sigma_i = \langle \rho_i, C_i\rangle$ are such that $\mathit{lab}[\![C_i]\!] \in \mathfrak{I}[\![P]\!]$, while μ^i are the portions (even empty) of trace of σ that are unchanged by opaque predicate insertion, that is $\forall \langle \rho, C\rangle \in \mu^i : \mathit{lab}[\![C]\!] \notin \mathfrak{I}[\![P]\!]$. By hypothesis η is obtained from σ by opaque predicate insertion, therefore η has the following structure: $\eta = \mu^1\eta_i^a\eta_i^b\mu^2\eta_j^a\eta_j^b\mu^3...\mu^l$, where $|\eta| = |\sigma| + |\mathfrak{I}[\![P]\!] \cap \{\mathit{lab}[\![C]\!] \mid \langle \rho, C\rangle \in \sigma\}|$ and $\eta_i^a\eta_i^b = \langle \rho_i, L_i : P^T \to \tilde{L}_i\rangle\langle \rho_i, \tilde{L}_i : A_i \to L_{i+1}\rangle$. Hence, following the definition of d_{OP} we have:

$$\begin{aligned} d_{OP}(\eta) &= d_{OP}(\mu^1\eta_i^a\eta_i^b\mu^2\eta_j^a\eta_j^b\mu^3...\mu^l) \\ &= \mu^1 d_{OP}(\eta_i^a\eta_i^b\mu^2\eta_j^a\eta_j^b\mu^3...\mu^l) \\ &= \mu^1\sigma_i d_{OP}(\mu^2\eta_j^a\eta_j^b\mu^3...\mu^l) \\ &= \mu^1\sigma_i\mu^2 d_{OP}(\eta_j^a\eta_j^b\mu^3...\mu^l) \\ &= ... = \mu^1\sigma_i\mu^2\sigma_j\mu^3...\mu^l = \sigma \end{aligned}$$

Thus, we have that $d_{OP}(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]) = d_{OP}(\{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in S^+[\![P]\!]\}) = \{d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) \mid \sigma \in S^+[\![P]\!]\} = \{\sigma \mid \sigma \in S^+[\![P]\!]\} = S^+[\![P]\!]$.

□

Observe that, by computing transformation d_{OP} on the obfuscated program semantics $S^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]$, and then deriving the corresponding program through $\mathbb{p}^+$, we obtain exactly the original program P, as shown in Figure 5. This means that, knowing the set OP an attack can eliminate the inserted opaque predicates, namely $\mathbb{p}^+ \circ d_{OP} \circ S^+$ acts as a deobfuscation technique. Thus, the insertion of true opaque predicates from set OP is not resilient with respect to an attacker that is able to detect the opaque predicates in OP.

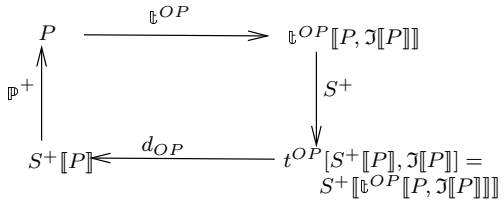

Figure 5: $\mathbb{p}^+ \circ d_{OP} \circ S^+$ is a deobfuscation technique

Example 2 *Let us consider the trace semantics $S^+[\![P]\!]$ of program P and a trace $\sigma \in S^+[\![P]\!]$. Let $\sigma = \langle \rho_0, C_0\rangle\langle \rho_1, C_1\rangle\langle \rho_2, C_2\rangle\langle \rho_3, C_3\rangle\langle \rho_4, C_4\rangle$, where $C_i = L_i : A_i \rightarrow L_{i+1}$. Assume $\mathfrak{I}[\![P]\!] = \{L_1, L_3\}$. The transformed trace is given by: $t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] = \langle \rho_0, C_0\rangle\langle \rho_1, L_1 : P^T \rightarrow \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : A_1 \rightarrow L_2\rangle\langle \rho_2, C_2\rangle\langle \rho_3, L_3 : P^T \rightarrow \tilde{L}_3\rangle \langle \rho_3, \tilde{L}_3 : A_3 \rightarrow L_4\rangle\langle \rho_4, C_4\rangle$. Clearly, $d_{OP}(\sigma) = \sigma$ and $d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) = d_{OP}(\langle \rho_0, C_0\rangle\langle \rho_1, L_1 : P^T \rightarrow \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : A_1 \rightarrow L_2\rangle\langle \rho_2, C_2\rangle \langle \rho_3, L_3 : P^T \rightarrow \tilde{L}_3\rangle \langle \rho_3, \tilde{L}_3 : A_3 \rightarrow L_4\rangle\langle \rho_4, C_4\rangle) = \langle \rho_0, C_0\rangle\langle \rho_1, C_1\rangle\langle \rho_2, C_2\rangle \langle \rho_3, C_3\rangle\langle \rho_4, C_4\rangle = \sigma$.*

Transformation d_{OP} is clearly additive and can therefore be viewed as an abstraction function. It is interesting to observe that, considering the concretization γ_{OP} induced by this abstraction, the property $\gamma_{OP} \circ d_{OP}$ corresponds to the most concrete property preserved by t^{OP}. In fact, knowing OP, the closure $\gamma_{OP} \circ d_{OP}$ observes traces up to opaque predicate insertion, which corresponds to the observation done by $\delta_{t^{OP}}$. In particular, given an obfuscated set of traces $\mathcal{X}$, the deobfuscation $d_{OP}(\mathcal{X}) = \mathcal{Y}$ eliminates the opaque predicates from traces in $\mathcal{X}$, and the concretization $\gamma_{OP}(\mathcal{Y})$ returns the set of all traces that can be obtained from traces in $\mathcal{Y}$ by opaque predicate insertion. This means that requiring $\mathcal{X}$ to be a fixpoint of $\gamma_{OP} \circ d_{OP}$, i.e., $\gamma_{OP}(d_{OP}(\mathcal{X})) = \mathcal{X}$, is equivalent to require that $\mathcal{X}$ satisfies $Pres_{P,t^{OP}}(\mathcal{X})$.

Theorem 8 *$\gamma_{OP} \circ d_{OP} \in uco(\wp(\Sigma^+))$ and $\gamma_{OP} \circ d_{OP} = \delta_{t^{OP}}$.*

PROOF: Function d_{OP} is clearly additive, and $\gamma_{OP} \circ d_{OP} \in uco(\wp(\Sigma^+))$. From Theorem 7 we have that property $\gamma_{OP} \circ d_{OP}$ is preserved by t^{OP}, i.e., $\gamma_{OP}(d_{OP}(S^+[\![P]\!])) = \gamma_{OP}(d_{OP}(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]))$, let us show that it coincides with $\delta_{t^{OP}}$. To do this we have to prove that, given $\mathcal{X} \in \wp(\Sigma^+)$: $\mathcal{X} = \gamma_{OP}(d_{OP}(\mathcal{X}))$ iff for every program $P \in \mathbb{P}$: $Pres_{P,t^{OP}}(\mathcal{X}) = true$.

($\Rightarrow$) By definition $\gamma_{OP}(d_{OP}(\mathcal{X})) = \{\sigma \mid d_{OP}(\sigma) \subseteq d_{OP}(\mathcal{X})\} = \{\sigma \mid \exists \eta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\eta)\}$. Thus, we have to prove that, for a given program P it holds that

$S^+[\![P]\!] \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$ iff $\forall \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!] : t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$. On the one hand, when $S^+[\![P]\!] \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$, then $S^+[\![P]\!] \subseteq \{\sigma \mid \exists \eta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\eta)\}$. This means that $\forall \sigma \in S^+[\![P]\!] : \exists \eta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\eta)$. Following the definition of t^{OP}, given the set $\mathfrak{I}[\![P]\!]$ of points candidate for opaque predicate insertion, we have $t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] = \{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in S^+[\![P]\!]\}$. Observe that $\forall \sigma \in S^+[\![P]\!] : t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\sigma))$, since we have shown that $d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) = d_{OP}(\sigma) = \sigma$. This means that $\forall \sigma \in S^+[\![P]\!] : \exists \eta \in \mathcal{X} : t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\sigma)) = \gamma_{OP}(d_{OP}(\eta)) \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$. Therefore $\forall \sigma \in S^+[\![P]\!] : t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\mathcal{X}))$, meaning that $t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$. The above proof works for any set of labels $\mathfrak{I}[\![P]\!]$, thus $\forall \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!] : t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \mathcal{X}$. On the other hand, when $\forall \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!] : t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$, then $\forall \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!] : t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \{\sigma \mid \exists \eta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\eta)\}$. We have shown that $\forall \mathfrak{I}[\![P]\!] \subseteq \mathit{lab}[\![P]\!] : d_{OP}(t^c[\mu, \mathfrak{I}[\![P]\!]]) = d_{OP}(\mu) = \mu$. Thus, $\forall \mu \in S^+[\![P]\!]$ for which there exists $\mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!]$ such that $t^{OP}[\mu, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\mathcal{X}))$ we have that $\mu \in \gamma_{OP}(d_{OP}(\mathcal{X}))$. Therefore, $\{\mu \mid t^{OP}[\mu, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\mathcal{X})), \mathfrak{I}[\![P]\!] \subseteq \mathit{lab}[\![P]\!]\} \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$, namely $S^+[\![P]\!] \subseteq \mathcal{X}$. The above reasoning is independent from the considered program P, which means that for any program $P \in \mathbb{P}$ we have that $Pres_{P,t^{OP}}(\mathcal{X}) = true$.

($\Leftarrow$) Assume that for all $P \in \mathbb{P}$: $Pres_{P,t^{OP}}(\mathcal{X}) = true$:

$\Rightarrow\ \forall P \in \mathbb{P} : S^+[\![P]\!] \subseteq \mathcal{X} \ \Leftrightarrow\ \forall \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!] : t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]] \subseteq \mathcal{X}$

$\Rightarrow\ \forall P \in \mathbb{P} : S^+[\![P]\!] \subseteq \mathcal{X} \ \Leftrightarrow\ \forall \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!] : \{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in S^+[\![P]\!]\} \subseteq \mathcal{X}$

$\Rightarrow\ \forall P \in \mathbb{P} : \forall \sigma \in \mathcal{X} : \{\eta \mid \eta = t^{OP}[\sigma, \mathfrak{I}[\![P]\!]], \mathfrak{I}[\![P]\!] \subseteq lab[\![P]\!]\} \subseteq \mathcal{X}$

$\Rightarrow\ \mathcal{X} = \{\eta \mid \exists \sigma \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\eta)\} = \gamma_{OP}(d_{OP}(\mathcal{X}))$

Hence, we have that $\delta_{t^{OP}} = \bigsqcup_{P \in \mathbb{P}} \{\mathcal{X} \mid Pres_{P,t^{OP}}(\mathcal{X})\} = \{\gamma_{OP}(d_{OP}(\mathcal{X})) \mid \mathcal{X} \in \wp(\Sigma^+)\} = \gamma_{OP}(d_{OP}(\wp(\Sigma^+)))$.

□

4.4 Detecting opaque predicates

It is clear that the efficiency of d_{OP} in eliminating true opaque predicates is based on the knowledge of OP. In fact, in the case of true opaque predicate insertion, the problem of deobfuscating a program reduces to the ability of detecting true opaque predicates. Let us recall that a true opaque predicate is a predicate that evaluates to *true* in every environment. Thus, understanding the presence of true opaque predicates in a program, means identifying those predicates that evaluate to *true* during every program execution. Given an obfuscated program $\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$ the set OP can be characterized by the following definition:

$$OP \stackrel{\text{def}}{=} \left\{ B \;\middle|\; \begin{array}{l} \exists C \in \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!] : \mathit{act}[\![C]\!] = B, \\ \forall \sigma \in S^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!], \forall \langle \rho, C \rangle \in \sigma : \\ (\mathit{act}[\![C]\!] = B) \Rightarrow (\mathbf{B}[\![B]\!]\rho = true) \end{array} \right\} \tag{3}$$

This means that by having access to the concrete semantics $S^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$ of the obfuscated program, which implies a precise evaluation $\mathbf{B}[\![B]\!]\rho$ of any test action B at any program point, we are able to construct the set OP that contains all the true opaque predicates that have been inserted in the program. Hence, if an attacker observes the concrete execution of an obfuscated program, it can deduce all the necessary information to deobfuscate it. In fact, opaque predicate insertion is an obfuscating transformation designed to confuse the control flow of a program and since program control flow

is an abstraction of trace semantics, we have that the obfuscation of the control flow may not cause confusion at the trace semantic level. This is the reason why, in order to better understand the obfuscating behaviour of opaque predicate insertion, we have to consider abstractions of trace semantics as we show in the following.

In Section 3.1 we have argued how attackers can be modeled as abstract interpretations of the concrete domain of computation of the maximal finite trace semantics of programs. In order to understand the potency and resilience of opaque predicate insertion we study what happens when the attackers have access only to the abstract semantics computed on their abstract domain. Let S^{φ} denote the abstract semantics computed by an attack $\varphi \in uco(\wp(\Sigma^+))$. In particular, if the concrete semantic is given by $S^+[\![P]\!] = lfpF^+[\![P]\!]$ then the abstract semantics is defined as $S^{\varphi}[\![P]\!] \stackrel{\text{def}}{=} lfpF^{\varphi}[\![P]\!]$, where F^{φ} is the best correct approximation of the concrete function F^+ on the abstract domain φ. We denote with $\hat{\mathfrak{E}}$ the set of abstract environments $\hat{\rho} : \mathbb{X} \to \varphi(\mathcal{D}_{\mathfrak{I}})$ that associates abstract values to program variables, with $\hat{\sigma}_i = \langle \hat{\rho}_i, C \rangle$ an abstract state, and with $\hat{\sigma}$ an abstract trace. Moreover, let $\varphi(\wp(\Sigma^+)) = \wp(\hat{\Sigma}^+)$ be the powerset of abstract traces. It is clear that, in this setting, the most powerful attacker is the one who has access to the most precise description of program behaviour, namely the one that is precise enough to compute the (concrete) program trace semantics $S^+[\![P]\!]$.

In general, the set OP^{φ} of true opaque predicates that an attacker modeled by abstraction φ is able to identify can be characterized as follows:

$$OP^{\varphi} \stackrel{\text{def}}{=} \left\{ B \;\middle|\; \begin{array}{l} \exists C \in \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!] : \mathit{act}[\![C]\!] = B, \\ \forall \hat{\sigma} \in S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!], \forall \langle \hat{\rho}, C \rangle \in \hat{\sigma} : \\ (\mathit{act}[\![C]\!] = B) \Rightarrow (\mathbf{B}^{\varphi}[\![B]\!]\hat{\rho} = \mathit{true}) \end{array} \right\} \tag{4}$$

Where $\mathbf{B}^{\varphi}[\![B]\!]\hat{\rho}$ denotes the abstract evaluation of the boolean expression B in the abstract environment $\hat{\rho}$. It is clear that, in general, the set of predicates classified as opaque by observing the abstract semantics S^{φ} is different from the set of predicates classified as opaque by observing program trace semantics S^+, i.e., $OP^{\varphi} \neq OP$. There are two causes of imprecision, both due to the loss of information implicit in the abstraction process:

- On the one hand, it may happen that φ is not powerful enough to recognize the constantly true value of some opaque predicates, namely there may exist an opaque predicate P^T such that $P^T \in OP$ while $P^T \notin OP^{\varphi}$ (see [19] for an example).
- On the other hand, an attack may classify a predicate as a true opaque predicate while it is not, namely there may exist a predicate Pr such that $Pr \in OP^{\varphi}$ while $Pr \notin OP$ (see Section 4.5 for an example).

The deobfuscation process that an attack φ can perform is expressed by the function $d_{OP^{\varphi}} : \wp(\hat{\Sigma}^+) \to \wp(\hat{\Sigma}^+)$, operating on abstract traces and on set OP^{φ} of opaque predicates.

$$d_{OP^{\varphi}}(\hat{\mathcal{X}}) \stackrel{\text{def}}{=} \{d_{OP^{\varphi}}(\hat{\sigma}) \mid \hat{\sigma} \in \hat{\mathcal{X}}\} \quad d_{OP^{\varphi}}(\hat{\sigma}) \stackrel{\text{def}}{=} \epsilon \; d_{OP^{\varphi}}(\hat{\sigma})$$

$d_{OP^{\varphi}}(\langle \hat{\rho}, C \rangle \langle \hat{\rho}', C' \rangle \hat{\eta}) \stackrel{\text{def}}{=}$

$$\begin{cases} \langle \hat{\rho}, C \rangle \; d_{OP^{\varphi}}(\langle \hat{\rho}', C' \rangle \hat{\eta}) & \text{if } \mathit{act}[\![C]\!] \notin OP^{\varphi} \\ d_{OP^{\varphi}}(\langle \hat{\rho}, \mathit{lab}[\![C]\!] : \mathit{act}[\![C']\!] \to \mathit{suc}[\![C']\!] \rangle \hat{\eta}) & \text{if } \mathit{act}[\![C]\!] \in OP^{\varphi} \end{cases}$$

We have that, in general, $OP \neq OP^{\varphi}$, and therefore that $S^{\varphi}[\![P]\!] \neq d_{OP}(S^{\varphi}[\![P]\!]) \neq d_{OP^{\varphi}}(S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$, where the first inequality follows by the fact that $d_{OP^{\varphi}}$ might eliminate a predicate Pr even if it is not opaque, i.e., when $Pr \in OP^{\varphi}$ while $Pr \notin OP$, and the second inequality by the fact that $d_{OP^{\varphi}}$ might not eliminate a predicate P^T that is opaque, i.e., when $P^T \notin OP^{\varphi}$ while $P^T \in OP$. Therefore, when $OP \neq OP^{\varphi}$ we have that attacker φ is not able to deobfuscate $\mathbb{t}^{OP}$. When an attack φ is not able to disclose the inserted opaque predicates, namely when $S^{\varphi}[\![P]\!] \neq S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$, the attack φ is defeated by the obfuscation (otherwise states the obfuscation is potent with respect to attack φ). This leads to the following definition of transformation potency:

Definition 7 *Transformation $\mathbb{t} : \mathbb{P} \rightarrow \mathbb{P}$ is potent with respect to an attack $\varphi \in uco(\wp(\Sigma^+))$ if there exists $P \in \mathbb{P}$ such that $S^{\varphi}[\![P]\!] \neq S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$.*

It is clear that the above definition of transformation potency is based on the abstract semantics computed by the attack and not on the abstraction of the concrete semantics as given in Definition 4 (where a transformation is potent if there exists an abstraction $\varphi \in uco(\wp(\Sigma^+))$ such that $\varphi(S^+[\![P]\!]) \neq \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])$). The two proposed definitions of transformation potency are deeply different and orthogonal. In fact, the results obtained in Section 3 referring to Definition 4, cannot be projected using Definition 7 of potency. However, the two definitions are both useful in understanding the obfuscating behaviour of program transformations. On the one hand, Definition 4 can be successfully applied to those obfuscation that have sensible effects on the concrete program semantics, namely those transformations that cannot be recovered by simply observing the concrete semantics of the obfuscated program (e.g., array merging, variable renaming, substitution of equivalent sequences of instructions). On the other hand, Definition 7 captures the obfuscating behaviour of program transformations that do not cause significant effects on the concrete semantics and that can be recovered by observing the concrete program semantics (e.g., opaque predicate insertion, code transportation, semantic nop insertion).

We are interested here in the study of the insertion of true opaque predicates and of the ability of attackers to recover the original program. In particular, it would be interesting to provide a formal characterization of the family of attackers that are able to disclose a given set of opaque predicates. Thus, given a set OP of true opaque predicates, we want to characterize the class of attacks φ such that $d_{OP^{\varphi}}(S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]) = d_{OP^{\varphi}}(S^{\varphi}[\![P]\!]) = S^{\varphi}[\![P]\!]$. Observe that this equality holds only when attack φ precisely identifies the set of inserted opaque predicates, namely when $OP = OP^{\varphi}$. When this happens we have that the obfuscation is harmless with respect to attack φ, namely that the insertion of true opaque predicates from OP is not able to obstruct attack φ. In the following we provide a characterization of the family of attacks able to disclose an interesting class of numerical opaque predicates.

4.5 Attacks and completeness

In [8] Collberg et al. observe that the study of random Java programs reveals that most predicates are extremely simple. In particular, common patterns include the comparison of integer quantities using binary operators such as equal to, greater than, smaller than, etc. It is clear that, in order to design stealthy obfuscating transformations, the inserted opaque predicates have to resemble the structure of predicates typically present in a program. For this reason we restrict out study to numerical opaque predicates on

$$\begin{array}{rl} \forall x, y \in \mathbb{Z}: & 7y^2 - 1 \neq x \\ \forall x \in \mathbb{Z}: & mod(3 \cdot 7^{4x-2} + 5 \cdot 4^{2x-1} - 5, 14) = 0 \\ \forall x \in \mathbb{Z}: & \sum_{i=1,2mod(i,2)\neq 0}^{2x-1} i = x \end{array}$$

Table 4: Commonly used opaque predicates [1]

integer values. In general, an opaque predicate of this kind is a function $\mathbb{Z}^n \rightarrow \mathcal{B}_3$ that takes an array of n integer values and returns $true$, $false$, $\bot$ or $\top$. A wide class of numerical opaque predicates can be characterized by the following structure:

$$\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) \ \mathbf{compare} \ g(\bar{x})$$

where **compare** stands for any binary operator in the set $\{=, \geq, \leq\}$, $\bar{x}$ is an array of n integer values, namely $\bar{x} \in \mathbb{Z}^n$, h and g are two functions over integers, in particular $h, g : \mathbb{Z}^n \rightarrow \mathbb{Z}$ (see Table 4 for some commonly used opaque predicates). Let us assume that each variable of program P ranges over $\mathbb{Z}$. Let $|var[\![P]\!]| = m$, then each abstraction (attack) $\varphi \in uco(\wp(\mathbb{Z}^m))$ induces an abstraction on the values of variables and therefore on the value that the opaque predicate input can assume. From now on, the abstract domain $\varphi \in uco(\wp(\mathbb{Z}^n))$ models the attack that observes an approximation φ of opaque predicate inputs. Let us consider a numerical opaque predicate of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, which verifies whether two functions h and g always return the same value when applied to the same array of integer values. In order to precisely detect the opaqueness of $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, one needs to check the *concrete test*, denoted as $CT^{h,g}$ and defined as follows:

$$CT^{h,g} \stackrel{\text{def}}{=} \forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$$

Once again, the set of predicates that satisfy the concrete test corresponds to the set OP of predicates characterized by equation (3). Our goal is to characterize the family of abstractions of $\wp(\mathbb{Z}^n)$ that perform the test of opaqueness for h and g in a precise way, namely the set of abstractions that loose information that is irrelevant for the precise computation of h and g. We are therefore interested in the family of abstract domains that are able to precisely compute functions h and g, which corresponds to the class of attacks able to deobfuscate the insertion of predicates of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$. Given $X \subseteq \mathbb{Z}^n$ let us consider the point to point definition of equality, where $h(X) \doteq g(X)$ if and only if $\forall \bar{x} \in X : h(\bar{x}) = g(\bar{x})$. Let $AT^{h,g}_{\varphi}$ denote the *abstract test* for opaqueness associated to an attack modeled by the abstract domain φ. The abstract test is defined as follows:

$$AT^{h,g}_{\varphi} \stackrel{\text{def}}{=} \forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\bar{x}))) \doteq \varphi(g(\varphi(\bar{x})))$$

Also in this case the set of opaque predicates satisfying the abstract test on φ corresponds to the set OP^{φ} of predicated characterized by equation (4). Once again, the precision of the abstract test strongly depends on the considered abstract domain. In particular, we have that an abstract test is sound when the satisfaction of the abstract test implies the satisfaction of the concrete one, and complete when the converse holds.

Definition 8 *Given an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, and an abstraction $\varphi \in uco(\wp(\Sigma^+))$, we say that:*

- *$AT^{h,g}_{\varphi}$ is sound when $AT^{h,g}_{\varphi} \Rightarrow CT^{h,g}$*

- *AT^{h,g_φ} is complete when $CT^{h,g} \Rightarrow AT^{h,g}_{\varphi}$*

When the abstract test $AT^{h,g}_{\varphi}$ is both sound and complete, i.e., $AT^{h,g}_{\varphi} \Leftrightarrow CT^{h,g}$, we say that attack φ *breaks* the opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$. In fact, in this case, the set of true opaque predicates coincides with the set of opaque predicates classified as opaque by the abstract test, meaning that we have obtained the desired equality $OP = OP^{\varphi}$.

It is possible to prove that when considering opaque predicates of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$ the abstract test defined above is complete for any attack $\varphi \in uco(\wp(\Sigma^+))$.

Theorem 9 *For any $\varphi \in uco(\wp(\Sigma^+))$ the abstract test $AT^{g,h}_{\varphi}$ is complete.*

PROOF: If the concrete test $CT^{h,g}$ is verified we have that $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, since $\varphi(\bar{x}) \subseteq \mathbb{Z}^n$ then $\forall \bar{x} \in \mathbb{Z}^n : \forall \bar{y} \in \varphi(\bar{x}) : h(\bar{y}) = g(\bar{y})$. This means that $\forall \bar{x} \in \mathbb{Z}^n : h(\varphi(\bar{x})) \doteq g(\varphi(\bar{x}))$, thus $\forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\bar{x}))) \doteq \varphi(g(\varphi(\bar{x})))$ that corresponds to the satisfaction of the abstract test $A^{g,h}_{\varphi}$.

□

This means that if a predicate is opaque then the attack recognises it, namely $OP \subseteq OP^{\varphi}$. Thus, $d_{OP^{\varphi}}(S^{\varphi}[\![P]\!]) = d_{OP^{\varphi}}(S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$. In fact, $d_{OP^{\varphi}}$ eliminates all the opaque predicates from the right term and the common regular predicate that are erroneously classified as opaque from both terms. For the same reason we have $S^{\varphi}[\![P]\!] \neq d_{OP^{\varphi}}(S^{\varphi}[\![P]\!])$. This means that $S^{\varphi}[\![P]\!] \neq d_{OP^{\varphi}}(S^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$ and therefore that φ is defeated by $\mathbb{t}^{OP}$. As argued above, attack φ is able to break the insertion of true opaque predicates when $OP = OP^{\varphi}$, which is ensured when the abstract test $AT^{h,g}_{\varphi}$ is both sound and complete. Theorem 9 guarantees the completeness of the abstract test, thus, in order to break an opaque predicate, we need to verify the soundness condition. In general $AT^{h,g}_{\varphi}$ is not sound, but it is possible to show that soundness is guaranteed when the abstract domain φ modeling the attack is $\mathcal{F}$-complete for both functions h and g.

Theorem 10 *Given an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{X}) = g(\bar{x})$, and an attack modeled by $\varphi \in uco(\wp(\Sigma^+))$, if the abstraction φ is $\mathcal{F}$-complete for both functions h and g then $AT^{h,g}_{\varphi}$ is sound.*

PROOF: We have to prove that $AT^{h,g}_{\varphi} \Rightarrow CT^{h,g}$. If the abstract test $AT^{h,g}_{\varphi}$ holds then $\forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\bar{x}))) \doteq \varphi(g(\varphi(\bar{x})))$, namely $\forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\varphi(\bar{x})))) \doteq \varphi(g(\varphi(\varphi(\bar{x}))))$. The abstract domain φ is $\mathcal{F}$-complete by hypothesis, therefore $\forall \bar{x} \in \mathbb{Z}^n : h(\varphi(\varphi(\bar{x}))) \doteq g(\varphi(\varphi(\bar{x})))$, which is equivalent to $\forall \bar{x} \in \mathbb{Z}^n : h(\varphi(\bar{x})) \doteq g(\varphi(\bar{x}))$. By definition of $\doteq$ this means that $\forall \bar{x} \in \mathbb{Z}^n : \forall \bar{y} \in \varphi(\bar{x}) : h(\bar{y}) = g(\bar{y})$. φ is extensive by hypothesis, namely $\bar{x} \in \varphi(\bar{x})$, and therefore $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, which corresponds to the satisfaction of the concrete test $CT^{h,g}$.

□

This means that when the abstract domain modeling the attack is able to precisely compute the functions composing the opaque predicate then the attack breaks the opaque predicate. Thus, given an attack φ and an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, the $\mathcal{F}$-completeness domain refinement of φ with respect to functions h and g adds the minimal amount of information to attack φ to make it able to defeat the considered opaque predicate. Hence, completeness domain refinement provides

here a systematic technique to design attacks that are able to break an opaque predicate of interest. Moreover, the completeness property of abstract interpretation precisely captures the ability of an attack to disclose an opaque predicate.

The above result holds also when considering $\leq, \geq$, and the corresponding point to point extensions $\dot{\leq}, \dot{\geq}$, instead of $=$ and $\doteq$.

Corollary 1 *Given an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x})$* **compare** *$g(\bar{x})$, and an attack $\varphi \in uco(\wp(\Sigma^+))$, if the abstraction φ is $\mathcal{F}$-complete for both functions h and g, then φ breaks opaque predicates that are instances of $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x})$ $\mathbf{compare}$ $g(\bar{x})$.*

In the following example we show how the lack of $\mathcal{F}$-completeness of the abstract domain modeling the attack can cause the abstract test to hold, even if the concrete one fails.

Example 3 *Let us consider the predicate $\forall x \in \mathbb{Z} : 2x^2 = 2x$, where $h(x) = 2x^2$ and $g(x) = 2x$. It is clear that $CT^{h,g}$ does not hold, since the predicate is not opaque. Let us consider an attack modeled by the abstract domain of $Parity = \{\top, \bot, even, odd\}$. In turns out that $AT^{h,g}_{Parity}$ holds, in fact:*

$$\begin{aligned} even &:: Parity(h(even)) = even = Parity(g(even)) \\ odd &:: Parity(h(odd)) = even = Parity(g(odd)) \end{aligned}$$

The reason way the abstract test holds on $Parity$ is the fact that $Parity$ is not $\mathcal{F}$-complete for both h and g. In fact, let $Parity = \gamma \circ \alpha$, then $2(\gamma(even)) = \{2x \mid x \in 2\mathbb{Z}\}$ which is strictly contained in $\gamma(2even) = \gamma(even) = 2\mathbb{Z}$. When computing the $\mathcal{F}$-completeness domain refinement of $Parity$ with respect to h and g, we close the considered abstract domain with respect to h and g. This means that, for example the elements $Double_2$, such that $\gamma(Double_2) = \{2x \mid x \in 2\mathbb{Z}\}$, $Double_1$, such that $\gamma(Double_1) = \{2x \mid x \in 2\mathbb{Z}+1\}$, $DoubleSq_2$, such that $\gamma(DoubleSq_2) = \{2x^2 \mid x \in 2\mathbb{Z}\}$, and $DoubleSq_1$, such that $\gamma(DoubleSq_1) = \{2x^2 \mid x \in 2\mathbb{Z}+1\}$, belong to $\mathcal{R}^{\mathcal{F}}_{h,g}(Parity) = Parity^+$. Observe that on this domain the abstract test does not hold any more, in fact $Parity^+(h(even)) = DoubleSq_2 \neq Double_2 = Parity^+(g(even))$, and so on for all the other elements since the direct image of all elements under h and g are precisely expressed by the domain obtained through the completeness refinement.

Comparing attacks

The completeness result obtained above allows us to compare on the lattice of abstract interpretation both the efficiency of different attacks in disclosing a particular opaque predicate, and the resilience of different opaque predicates with respect to an attack.

Let us consider a predicate $P^T : \forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, and let us denote with $\mathcal{R}_{P^T}$ the completeness domain refinement needed to make an attack able to break P^T. Let $Potency(P^T, \varphi)$ denote the potency of opaque predicate P^T with respect to attack φ, and $Resilience(P^T, \varphi)$ the resilience of opaque predicate P^T in preventing attack φ.

Definition 9 *Given two attacks $\varphi, \psi \in uco(\wp(\Sigma^+))$ and two opaque predicates P^T_1 and P^T_2, we have that:*

- *when $\varphi \sqsubset \psi$ and $\mathcal{R}_{P^T_1}(\psi) = \mathcal{R}_{P^T_1}(\varphi)$ we say that $Potency(P^T_1, \psi)$ is greater than $Potency(P^T_1, \varphi)$*

- *while, when* $\mathcal{R}_{P_1^T}(\varphi) \sqsubset \mathcal{R}_{P_2^T}(\varphi)$*, we say that* $Resilience(P_1^T, \varphi)$ *is greater than* $Resilience(P_2^T, \varphi)$

The first point of the above definition refers to the situation presented in Figure 6(a), where $\varphi \sqsubset \psi$ and $\mathcal{R}_{P_1^T}(\psi) = \mathcal{R}_{P_1^T}(\varphi)$. In this case we have that predicate P_1^T is more potent with respect to attack ψ than with respect to attack φ. In fact, more information needs to be added to ψ than to φ in order to gain an attack able to break P_1^T, namely ψ is more "far" than φ in disclosing P_1^T. The same reasoning allows us to compare

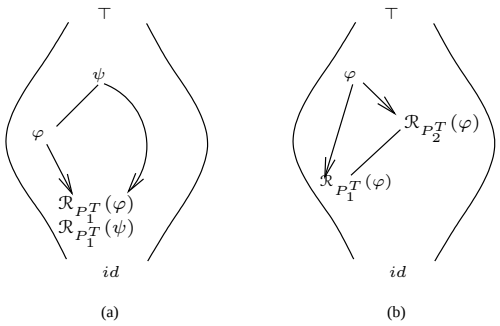

Figure 6: Comparing attacks

the resilience of different opaque predicates in the lattice of abstract interpretation. In fact, the second point of the above definition considers two predicates P_1^T and P_2^T and an attack $\varphi \in uco(\wp(\Sigma^+))$. and assumes that $\mathcal{R}_{P_1^T}(\varphi) \sqsubset \mathcal{R}_{P_2^T}(\varphi)$ as shown in Figure 6(b). In this case we can say that the insertion of opaque predicate P_1^T is more efficient in obstructing attack φ than the insertion of opaque predicate P_2^T, since more information needs to be added to φ in order to disclose P_1^T than P_2^T. Thus, a possible way to understand which opaque predicate in OP is more efficient in preventing a given attack φ, it is to compute the fixpoint solution of the completeness domain refinement of φ with respect to the different opaque predicates available, and then choose the one that corresponds to the most concrete refinement. In fact, the closer the refined attack is to the identical abstraction (concrete semantics), the higher is the resilience of the opaque predicate. In particular, if $\mathcal{R}_{P^T}(\varphi) = id$, it means that the attack φ can break the considered opaque predicate only if it can access the concrete program semantics. In this case the considered opaque predicate provides the best obstruction to φ.

5 Discussion

In order to fulfill the lack of a theoretical basis for code obfuscation, we have proposed a formal approach to code obfuscation based on program semantics and abstract interpretation. The key idea of our approach is to model attacks as abstract domains, where the abstraction encodes the power of the attack, namely what the attack can observe of program execution. In fact, the proposed semantic framework relies on a semantics-based definition of code obfuscation and on an abstract interpretation-based model for attacks. In particular, we characterize the obfuscating behaviour of a program transformation $\mathbb{t}$ in terms of the most concrete semantic property $\delta_{\mathbb{t}}$ it preserves, namely in terms of the most powerful attack for which the obfuscation is harmless. In fact, given a transformation $\mathbb{t}$, property $\delta_{\mathbb{t}}$ precisely expresses the amount of information

still available after the obfuscation $\mathbb{t}$, namely what the obfuscated program might reveal about the original program. In this setting, any program transformation $\mathbb{t}$ can be seen as an obfuscator that is potent with respect to any attacker modeled by an abstract domain φ that is not preserved by $\mathbb{t}$. In particular, the semantics-based notion of potency given in Definition 4 states that a transformation $\mathbb{t}$ is potent if it defeats attacks modeled as properties of program trace semantics, namely if there exists a property $\varphi \in uco(\wp(\Sigma^+))$ such that $\varphi(S^+[\![P]\!]) \neq \varphi(S^+[\![\mathbb{t}[\![P]\!]]\!])$. This measure of potency fits transformations that deeply modify program trace semantics, namely that modify program behaviour in a way that is noticeable and not trivially undone by an attacker that observes program trace semantics. Moreover, this notion of transformation potency provides an advanced technique for comparing obfuscating algorithms relative to their potency in the lattice of abstract interpretation (as stated by Definition 6). Among the existing obfuscating transformations whose potency can be modeled by this definition we mention: the substitution of equivalent sequences of commands, variable renaming and data obfuscations such as splitting and merging arrays. In fact, these obfuscations modify the structure of program trace semantics in a sensible way: replacing equivalent sequences of commands implies a modification of the program execution traces, and the renaming of variables and the splitting and merging of arrays cause a modification of every program state whose command uses a renamed variable or an obfuscated array. Thus, the potency of these obfuscations can be captured by Definition 4 of transformation potency.

However, Definition 4 is not adequate for modeling the potency of obfuscating transformations that cause only minor changes to the program trace semantics, namely that do not confuse an attack that has access to the trace semantics of the obfuscated code, as in the case of opaque predicate insertion. In this case, for example, we need a notion of program potency that captures the *noise* introduced at the level of program control flow, which is an abstraction of trace semantics. This observation has led to Definition 7, where transformation potency is formalized with respect to the abstract semantics computed on the abstract domain modeling the attack; a transformation $\mathbb{t}$ is potent if there exists an abstraction φ such that $S^{\varphi}[\![P]\!] \neq S^{\varphi}[\![\mathbb{t}[\![P]\!]]\!]$. This definition can model the potency of several existing obfuscating techniques: opaque predicate insertion, control flow flattening, loop unrolling and semantic nop insertion. Control flow flattening and loop unrolling are control code obfuscations that, like opaque predicate insertion, try to mask the control flow of the original program. Once again, in order to notice the obscurity added at the control flow level by these transformations, we need to consider the abstract semantics computed on the abstract domain modeling the attackers. Moreover, as in the case of opaque predicate insertion, when dealing with semantic nop insertion we have that an attack is confused by the insertion of semantic nops only when it is not able to recognize the inserted semantic nops. Also in this case, the ability of an attacker in identifying the inserted semantic nops might be expressed in terms of the precision of the abstract domain modeling the attack.

It is clear that the two definitions of potency are deeply different and orthogonal and that each of them fits different kinds of obfuscations. In Section 4.4, we have seen in detail how Definition 7 of transformation potency properly models an obfuscation that inserts true opaque predicates, from which we can deduce that it is appropriate also for modeling the insertion of false opaque predicates. Moreover, it is reasonable to assume that also the potency of transformations that insert correlated opaque predicates and distributed opaque predicates can be modeled by Definition 7. In this case the opaqueness of the predicates ensures that only the correct paths are executed, while confusion can be inserted in the "fake" paths. In this setting, an attack is able to disclose

a set of correlated opaque predicates only if it is able to understand that there exists a relation between the values of these predicates during execution. Thus, it seems that in order to disclose correlated opaque predicates an attacker should be precise for some sort of relational analysis.

In the particular case of the insertion of true opaque predicates, the use of abstract interpretation ensures that, when the abstraction is complete, the attack is able to break the opaque predicate and to remove the obfuscation. This proves that deobfuscation in the case of opaque predicates requires complete abstractions and therefore that the potency and resilience of opaque predicates can be measured by the amount of information that has to be added to the incomplete domain to become complete. This allows us to compare both the potency of different opaque predicates with respect to a given attack, and the resilience of an opaque predicate with respect to different attacks. Some further work is necessary in order to validate our theory in practice. In fact, while measuring the resilience of opaque predicates in the lattice of abstract domains may provide an absolute and domain-theoretical taxonomy of attacks and obfuscators, it would be interesting to investigate the true effort, in terms of dynamic testing, which is necessary to enforce 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. Preliminary work in this direction shows promising experimental results, as described in [19].

Another interesting field that commonly uses code obfuscation is the one of "biologically inspired diversity". In this setting, obfuscating transformations are used to generate many different versions of the same program in order to prevent malware infection [21, 38]. In fact, machines that execute the same programs are likely to be vulnerable to the same attacks. Malware exploit vulnerabilities in order to propagate and perform their damage, meaning that all the systems sharing the same configuration will be susceptible to the same malware attacks. On the other hand, different versions of the same program are less prone to having vulnerabilities in common. This means that diverse versions of the same program will make malware infection and propagation much harder. In this setting, it would be interesting to see if our theoretical framework for code obfuscation could be used to better understand and formalize the level of security that program diversity guarantees.

Acknowledgments

This work has been partially supported by the FIRB grant "Abstract Interpretation and Model Checking for the verification of embedded systems", and the MUR-COFIN grant AIDA. The results present in this work are an extended and reviewed version of [17, 18].

References

[1] G. Arboit. A method for watermarking Java programs via opaque predicates. In *Proc. Int. Conf. Electronic Commerce Research* (*ICECR-5*), 2002.

[2] B. Barak, O. Goldreich, R. Impagliazzo, and S. Rudich. On the (im)possibility of obfuscating programs. In *Advances in Cryptology, Proc. of Crypto'01*, volume 2139 of *LNCS*, pages 1–18. Springer-Verlag, 2001.

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

[4] C. Collberg and K. Heffiner. The obfuscation executive. In *Proc. Information Security Conference* (*ISC'04*), volume 3225 of *LNCS*, pages 428–440, 2004.

[5] C. Collberg and C. Thomborson. Breaking abstractions and unstructural data structures. In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '98*), pages 28–37, 1998.

[6] C. Collberg and C. Thomborson. Software watermarking: models and dynamic embeddings. In *Principles of Programming Languages 1999*, (*POPL '99*), 1999.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.

[8] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* (*POPL '98*), pages 184–196. ACM Press, 1998.

[9] C. Consel and C. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM Symp. on Principles of Programming Languages* (*POPL '93*), pages 493–501. ACM Press, 1993.

[10] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

[11] P. Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. *Université Scientifique et Médicale de Grenoble*, 1978.

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

[13] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

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

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

[16] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, New York, NY, 2002.

[17] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 301–310, Koblenz, Germany, 2005.

[18] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proc. of the 32nd International Colloquium on Automata, Languages and Programming* (*ICALP '05*), volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336. Springer-Verlag, 2005.

[19] M. Dalla Preda, M. Madou, R. Giacobazzi, and K. De Bosschere. Opaque predicate detection by abstract interpretation. In *Proc. of the 11th International Conf. on Algebraic Methodology and Software Technology* (*AMAST '06*), volume 4019 of *LNCS*, pages 81–95. Springer-Verlag, 2006.

[20] J. Marciniak editor. *Encyclopedia of Software Engineering*. J. Wiley & Sons, Inc, 1994.

[21] S. Forrest, A. Somyaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hot Topics in Operating systems*, pages 67–72, 1997.

[22] 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.

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

[24] James R. Gosler. Software protection: myth or reality? In *Proc. Advances in Cryptology* (*CRYPTO'85*), pages 140–157, 1985.

[25] G. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, Basel, Switzerland, 1978.

[26] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[27] W. A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. In *SIGPLAN Notices*, volume 16, pages 63–74, 1981.

[28] N. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–504, 1996.

[29] G. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st ACM Symp. on Principles of Programming Languages* (*POPL '73*). ACM Press, 1973.

[30] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Computer Security Symposium* (*CSS '03*), pages 290–299, 2003.

[31] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *Proc. Internat. Workshop on Information Security Applications* (*WISA'05*), volume 3786 of *LNCS*, pages 194–206, 2005.

[32] A. Majumdar and C. Thomborson. Manufactoring opaque predicates in distributed systems for code obfuscation. In *Proc. 29th Australasian Computer Science Conference (ACSC'06)*, volume 48 of *CRPIT*, pages 187–196, 2006.

[33] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEEE Trans. Fundamentals*, E86-A(1), 2003.

[34] E. I. Oviedo. Control Flow, Data Flow and Programmers Complexity. In *Proc. of COMPSAC 80*, pages 146–152. Chicago, IL, 1980.

[35] R. Paige. Future directions in program transformations. *ACM SIGPLAN Not.*, 32(1):94–97, 1997.

[36] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th IEEE Annual Security Applications Conference* (*ACSAC '00*), pages 308–316, 2000.

[37] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *STTT*, 2(2):192–201, 1998.

[38] R. Pucella and F.B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *Proceedings of the 19th IEEE Computer Security Foundation Workshop*, pages 230–241, 2006.

[39] S. K. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th. IEEE Working Conference on Reverse Egineering* (*WCRE '05*), 2005.

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

[41] M. Weiser. Program slicing. *IEEE Trans. Software Engineering*, 10(4):352–357, 1984.

[42] H. Yang and Y. Sun. Reverse engineering and reusing COBOL programs: A program transformation approach. In *IWFM '97 Electronic Workshop in Computing*, 1997.

6 Appendix

Syntactic opaque predicate insertion

Given the semantic transformation t^{OP}, defined in Section 4.1, that performs the insertion of true opaque predicates form the set OP, in the following we report the details of the derivation of the corresponding syntactic transformation $\mathbb{p}^+ \circ t^{OP} \circ S^+$.

Step 1: When we consider the program trace semantics expressed in fixpoint form, we have that $\mathbb{p}^+(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]])$, reduces to $\mathbb{p}^+(t^{OP}[lfpF^+[\![P]\!], \mathfrak{I}[\![P]\!]])$.

Step 2: Let us compute the transformation t^{OP} of the program semantics $S^+[\![P]\!]$ expressed in fixpoint form $lfpF^+[\![P]\!]$, in order to establish the local commutation property necessary for fixpoint transfer:

$t^{OP}[F^{+}[\![P]\!](\mathfrak{X}), \mathfrak{I}[\![P]\!]] = t^{OP}[\mathfrak{T}[\![P]\!] \cup \{ss'\sigma \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathfrak{X}\}, \mathfrak{I}[\![P]\!]] =$
$t^{OP}[\mathfrak{T}[\![P]\!], \mathfrak{I}[\![P]\!]] \cup t^{OP}[\{ss'\sigma \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathfrak{X}\}, \mathfrak{I}[\![P]\!]]$

Let us consider the two terms of the above union separately. For the first term we have:

$$t^{OP}[\mathfrak{T}[\![P]\!], \mathfrak{I}[\![P]\!]] = \{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in \mathfrak{T}[\![P]\!]\} =$$

$$\{t^{OP}[\langle \rho, L : A \to L' \rangle, \mathfrak{I}[\![P]\!]] \mid L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathfrak{L}[\![P]\!]\} =$$

$$\{\langle \rho, L : A \to L' \rangle \mid L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathfrak{L}[\![P]\!], L \notin \mathfrak{I}[\![P]\!]\} \ \cup$$
$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle | L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathfrak{L}[\![P]\!],$$
$$L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

Considering the second term, we have that:

$$t^{OP}[\{ss'\sigma \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathfrak{X}\}, \mathfrak{I}[\![P]\!]] =$$
$$\{t^{OP}[ss'\sigma, \mathfrak{I}[\![P]\!]] \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathfrak{X}\}$$

assuming $s = \langle \rho, L : A \to L' \rangle, s' = \langle \rho', C' \rangle$, we obtain:

$$\{\langle \rho, L : A \to L' \rangle t^{OP}[\langle \rho', C' \rangle \sigma, \mathfrak{I}[\![P]\!]] \mid \mathit{lab}[\![C']\!] = L', \rho' \in \mathbf{A}[\![A]\!]\rho,$$
$$L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \langle \rho', C' \rangle \sigma \in \mathfrak{X}, L \notin \mathfrak{I}[\![P]\!]\} \ \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle t^{OP}[\langle \rho', C' \rangle \sigma, \mathfrak{I}[\![P]\!]] \mid \mathit{lab}[\![C']\!] = L',$$
$$\rho' \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \langle \rho', C' \rangle \sigma \in \mathfrak{X}, L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

that, given $\sigma' = \langle \rho', C' \rangle \sigma$, reduces to:

$$\{\langle \rho, L : A \to L' \rangle t^{OP}[\sigma', \mathfrak{I}[\![P]\!]] \mid \mathit{lab}[\sigma'] = L', \mathit{env}[\sigma'] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P,$$
$$\rho \in \mathfrak{E}[\![P]\!], \sigma' \in \mathfrak{X}, L \notin \mathfrak{I}[\![P]\!]\} \ \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle t^{OP}[\sigma', \mathfrak{I}[\![P]\!]] \mid \mathit{lab}[\sigma'] = L', \mathit{env}[\sigma'] \in \mathbf{A}[\![A]\!]\rho,$$
$$L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \sigma' \in \mathfrak{X}, L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

then, assuming $\hat{\sigma} = t^{OP}[\sigma', \mathfrak{I}[\![P]\!]]$, we obtain:

$$\{\langle \rho, L : A \to L' \rangle \hat{\sigma} \mid \mathit{lab}[\hat{\sigma}] = L', \mathit{env}[\hat{\sigma}] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P,$$
$$\rho \in \mathfrak{E}[\![P]\!], \hat{\sigma} \in t^{OP}[\mathfrak{X}, \mathfrak{I}[\![P]\!]], L \notin \mathfrak{I}[\![P]\!]\} \ \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \hat{\sigma} \mid \mathit{lab}[\hat{\sigma}] = L', \mathit{env}[\hat{\sigma}] \in \mathbf{A}[\![A]\!]\rho,$$
$$L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \hat{\sigma} \in t^{OP}[\mathfrak{X}, \mathfrak{I}[\![P]\!]], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

where given a trace σ: $\mathit{env}[\sigma] = \mathit{env}[\sigma_0]$ and $\mathit{env}[\langle \rho, C \rangle] = \rho$, while $\mathit{lab}[\sigma] = \mathit{lab}[\sigma_0]$ and $\mathit{lab}[\langle \rho, C \rangle] = \mathit{lab}[\![C]\!]$. By defining $F^{OP}[\![P]\!](t^{OP}[\mathfrak{X}, \mathfrak{I}[\![P]\!]])$ as given by the union of the elements obtained by the above computation, we have:

$F^{OP}[\![P]\!](t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) \stackrel{\text{def}}{=}$

$$\{\langle \rho, L : A \to L' \rangle \mid L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathfrak{L}[\![P]\!], L \notin \mathfrak{I}[\![P]\!]\} \cup$$
$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \mid L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathfrak{L}[\![P]\!], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\} \cup$$
$$\{\langle \rho, L : A \to L' \rangle \hat{\sigma} \mid \mathit{lab}[\hat{\sigma}] = L', \mathit{env}[\hat{\sigma}] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \hat{\sigma} \in t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]], L \notin \mathfrak{I}[\![P]\!]\} \cup$$
$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \hat{\sigma} \mid \mathit{lab}[\hat{\sigma}] = L', \mathit{env}[\hat{\sigma}] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \hat{\sigma} \in t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

Thus, $t^{OP} \circ F^{+} = F^{OP} \circ t^{OP}$, and applying the fixpoint transfer theorem we have that $t^{OP}[\mathit{lfp} F^{+}[\![P]\!], \mathfrak{I}[\![P]\!]]$ can be expressed as $\mathit{lfp} F^{OP}[\![P]\!]$.

Step 3: Let us compute the abstraction $\mathbb{p}^{+}$ of $F^{OP}[\![P]\!]$ in order to verify the commutation property necessary for fixpoint transfer:

$\mathbb{p}^{+}(F^{OP}[t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]]) =$

$$\{\{L : A \to L'\} \mid L : A \to L' \in P, L' \in \mathfrak{L}[\![P]\!] L \notin \mathfrak{I}[\![P]\!]\} \cup$$
$$\{\{L : P^T \to \tilde{L}; \tilde{L} : A \to L'\} \mid L : A \to L' \in P, L' \in \mathfrak{L}[\![P]\!], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\} \cup$$
$$\{\{L : A \to L'\} \cup \mathbb{p}^{+}(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) \mid L : A \to L' \in P, L \notin \mathfrak{I}[\![P]\!], \exists C \in \mathbb{p}^{+}(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) : \mathit{lab}[\![C]\!] = L'\} \cup$$
$$\{\{L : P^T \to \tilde{L}; \tilde{L} : A \to L'\} \cup \mathbb{p}^{+}(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) \mid L : A \to L' \in P, L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New, \exists C \in \mathbb{p}^{+}(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) : \mathit{lab}[\![C]\!] = L'\}$$

Step 4: Defining $\mathbb{F}^{OP}[\![P]\!](\mathbb{p}^{+}(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]))$ as the union above, we have that $\mathbb{p}^{+} \circ F^{OP}[\![P]\!] = \mathbb{F}^{OP}[\![P]\!] \circ \mathbb{p}^{+}$, and therefore $\mathbb{p}^{+}(\mathit{lfp} F^{OP}[\![P]\!]) = \mathit{lfp} \mathbb{F}^{OP}[\![P]\!]$. From the definition of $\mathbb{F}^{OP}$ it is possible to derive an extended iterative algorithm that inserts opaque predicates.