Characterizing A Property-driven Obfuscation Strategy

Mila Dalla Preda [*], Isabella Mastroeni . [*]
Dipartimento di Informatica, University of Verona, Italy
E-mail: {mila.dallapreda,isabella.mastroeni}@univr.it

Abstract. In recent years, code obfuscation has attracted both researchers and software developers as a useful technique for protecting secret properties of proprietary programs. The idea of code obfuscation is to modify a program, while preserving its functionality, in order to make it more difficult to analyze. Thus, the aim of code obfuscation is to conceal certain properties to an attacker, while revealing its intended behavior. However, a general methodology for deriving an obfuscating transformation from the properties to conceal and reveal is still missing. In this work, we start to address this problem by studying the existence and the characterization of function transformers that minimally or maximally modify a program in order to reveal or conceal a certain property. Based on this general formal framework, we are able to provide a characterization of the maximal obfuscating strategy for transformations concealing a given property while revealing the desired observational behavior. To conclude, we discuss the applicability of the proposed characterization by showing how some common obfuscation techniques can be interpreted in this framework. Moreover, we show how this approach allows us to deeply understand what are the behavioral properties that these transformations conceal, and therefore protect, and which are the ones that they reveal, and therefore disclose.

Keywords: Program transformation, abstract interpretation, program semantics, code obfuscation.

1. Introduction

The last years have seen a considerable growth in the amount of software that is distributed over the Internet and in the number of wireless devices that dominate our society. Common classes of web applications that are part of our daily lives include e-mail clients, e-banking, e-commerce, social shopping, social networks and e-voting. In this complex scenario, users need to protect their devices against malicious software attacks (e.g., software viruses and internet worms), while software developers need to protect their products against malicious host attacks that usually aim at stealing, modifying or tampering the code in order to obtain (economic) advantages over it.

A key challenge, in defending code running on an untrusted host, is that there is no limit on the techniques that a human attacker can use to extract or modify sensitive data from the code and/or to violate its intellectual property. Indeed, software developers lose the control of their applications once they are distributed to a client machine. The most common malicious host attacks against proprietary programs

[*] This work has been supported by the MIUR FIRB project FACE (Formal Avenue for Chasing malwarE) RBFR13AJFT

are malicious reverse-engineering, software piracy and software tampering [5]. Malicious reverse engineering refers to those techniques inspecting the inner workings of software applications for unlawful purposes. Both software tampering and software piracy need a preliminary reverse-engineering phase in order to understand the inner working of programs they want to tamper with or to use, without authorization. Thus, the first defense against malicious host attacks consists in impeding reverse engineering as much as possible.

Of course there are legal measures to protect software against malicious host attacks, such as copyrights, patents and licenses. However, legal countermeasures are expensive and it may be hard, for a software developer, to enforce the law against a larger and more powerful competitor. For this reasons, software developers often resort to hardware and software solutions for protecting their products. Hardware solutions, for example by means of dongles, cryptographic processors or smart cards, are less flexible than software solutions, have higher costs and require more effort in maintenance. It is therefore important to develop general software security techniques, which are hardware and platform independent, and which are able to ensure the desired security requirements. Two of the most important software defense techniques against reverse engineering are code obfuscation and software watermarking. Code obfuscation [7] aims at transforming programs in order to make them more difficult to analyze, but anyway preserving their functionality, while software watermarking [6] inserts a signature in the code in order to link the software to the legitimate owner. Software watermarking is useful in identifying a violation once it has happened, while code obfuscation is considered a promising technique to prevent malicious attacks. In recent years, code obfuscation has attracted the attention of many researchers and this has led to the design and implementation of many obfuscation strategies (e.g., [7, 18, 33, 34, 40]) together with the study of theoretical models and frameworks for understanding the potentialities and limits of obfuscation (e.g., [2, 4, 16, 21, 25, 39]).

A well-known negative theoretical result on code obfuscation is the one of Barak et al. [2] showing that obfuscation is impossible. Note that, this result states the impossibility of an "ideal" obfuscator that obfuscates every program by revealing only the properties that can be derived from the I/O semantics. Indeed, the obfuscator of Barak et al. is a program transformer that: (1) preserves program's functionality, (2) generates obfuscated programs whose slowdown with respect to the original program is polynomial both in time and space, (3) and satisfies the virtual black box (VBB) property, namely anything that one can compute from the obfuscated program can also be computed having access to the input/output behavior of the original program. Given the impossibility of such a VBB-obfuscator, Barak et al. suggested another notion of obfuscation called indistinguishability obfuscator, that weakens the VBB property. Indeed, an indistinguishability obfuscator has to (1) preserve program's functionality, (2) cause a polynomial slowdown and (3) be such that it is not possible to distinguish the obfuscated version of two equivalent programs. Recently, Garg et al. [19] have provided the first construction for indistinguishability obfuscation, thus proving the existence of such an obfuscation. However, the proposed construction has a very high complexity that makes it impractical to use. Observe that, the results of Barak et al. and of Garg et al. aim at studying the existence of a VBB, or indistinguishability obfuscator, working for each program (more specifically for each polynomial size circuit) and guaranteeing a polynomial (in the worst case) loss of performance in time and space. Besides the negative result of Barak et al. and the positive, but still impractical, result of Garg et al., in recent decades, we have seen a big effort in developing and implementing new and efficient obfuscation strategies. Of course, these obfuscating techniques introduce a kind of practical obfuscators weakening the VBB-obfuscators in different ways, and which can be effectively used in real application protection in the markets. For example, these obfuscators may work only for a certain class of programs, or may be able to hide only certain properties of programs (e.g., control flow), or may cause a loss of performance in time and space that is more than polynomial. Indeed, the attention on code obfuscation poses the need to deeply understand *what* it is possible to obfuscate of a particular program, *when* it is possible to obfuscate it, and *how* it is possible to obfuscate it. Surely, an important task in this field, related to *what* we can obfuscate, consists in understanding obfuscations by characterizing which kind of attacks they are able to defeat. We believe that the development of a systematic strategy for the design of an obfuscator parameterized with respect to program properties both to protect (conceal) and to preserve (reveal), would be an important advance in the understanding of property-driven obfuscation techniques. In particular, it would provide a better insight into the relation between the property preserved by an obfuscator, which usually is the I/O program behavior but which can be any (more precise) observable property of the program, and the property protected, e.g., the program dependencies, the control structure, and so on. The first step toward the development of a systematic design of property-driven obfuscation strategy is the ability to formally, and therefore semantically, characterize the obfuscation of a particular program with respect to the property to be protected and preserved. The idea is that a program property is obfuscated, namely protected by obfuscation, when analyzing the obfuscated program it is not possible to observe precisely the property of the original program. This means that obfuscation has modified the program in order to add noise and make imprecise the observation of the property to protect. In this work we address the problem at the semantic level: we provide a methodology for designing semantic code transformations that act as obfuscation with respect to the properties to be protected and preserved. The necessity of fixing and considering both these constraints comes from the fact that, while transforming a code for protecting some of its properties, we could transform other code properties that we would like to preserve, such as the I/O functionality. For this reason, we need to formally fix also the property we aim at preserving, in order to provide a bound to the noise we can add for obfuscating the program semantics.

By considering, for instance, data-type obfuscation [18], the obfuscation is achieved by transforming data manipulations in order to deceive particular data observations. Without entering in details, consider the following code $\mathtt{P}$ and a possible data-type obfuscation of variable s concealing the parity property of the values assumed by variable s during computation and at the end of the program, while preserving its I/O value:

$$\mathtt{P} = \left[\begin{array}{l} {}^{1.}x := 2;\\ {}^{2.}s := 0;\\ {}^{3.}\textbf{while } x < 5 \textbf{ do}\\ \qquad {}^{4.}s := s + x;\\ \qquad {}^{5.}x := x + 2;\\ \qquad \textbf{endw}\\ {}^{6.}\textbf{output}(s); \end{array}\right. \qquad \text{and} \qquad \mathcal{O}(\mathtt{P}) = \left[\begin{array}{l} {}^{1.}\ x := 4;\\ {}^{2.}s := 0;\\ {}^{3.}\textbf{while } x < 10 \textbf{ do}\\ \qquad {}^{4.}s := s + x/2;\\ \qquad {}^{5.}x := x + 4;\\ \qquad \textbf{endw}\\ {}^{6.}\textbf{output}(s); \end{array}\right.$$

In this case, all the definitions of x are doubled, while in all the uses of x its value is divided by 2. In this way, the final value of s (computed in terms of x) does not change, but some analyses are confused. In particular, we can observe that if we statically analyze the parity of s, then in $\mathtt{P}$ we precisely conclude that s at statement 6 is odd, while in the obfuscated program $\mathcal{O}(\mathtt{P})$ we cannot conclude anything on the parity of s at statement 6, due to the computation in terms of the integer division $x/2$, whose parity is undetermined when we only know the parity of x. Indeed, we can observe that, we do confuse other program properties, such as the range of values for x (interval analysis [11]), but this is ignored since the two required constraints are satisfied (preserve the I/O of s and protect the parity of s). Note that, we could conceal the parity analysis of s also by changing only statement 4 (without changing statements 1 and 5), but in this way we would also change the I/O behavior, making this transformation not acceptable as obfuscation.

This example shows us that, in order to formally reason on what is protected, namely what is *concealed*, and what is preserved, namely what is *revealed*, by an obfuscating transformation, we need to focus on the effects that obfuscation has on the program semantics. In [14,16] it is shown that the semantic level allows us to understand the protection capabilities and the limits of an obfuscating strategy with respect to a particular program. We start from the approach of Dalla Preda and Giacobazzi in [14, 16] (then developed also in [20, 21]), where a formal framework for code obfuscation is provided by taking into account the effects that an obfuscating transformation has on trace semantics, and by modeling attackers as approximations of trace semantics. In order to reason on the semantic aspects of obfuscation, we refer to the formal framework introduced by Cousot and Cousot [12], where the relation between syntactic and semantic transformations is formalized in terms of abstract interpretation by considering programs as abstractions of their semantics. Indeed, the above mentioned semantics-based formalization of code obfuscation, has allowed us to qualitatively compare the efficiency of different obfuscating transformations, and in certain cases [13, 15], the semantic understanding of the obfuscation strategies has allowed us to design an efficient attacker who may be able to de-obfuscate the program, thus proving the potentialities and limits of the considered transformations.

In this work, we propose a general methodology for characterizing (when it exists) a property-driven obfuscating transformation that protects a given semantic property (*concealment transformer*), and preserves another semantic property (*revelation transformer*). In particular, we observe that a concrete example of revelation transformer is the program slicing transformation, which transforms a program looking for the maximal subprogram *preserving* the I/O behavior on the criterion variables [41]. On the other hand, the concealment transformer adds any computation that may conceal the property to protect, potentially concealing, in this way, also the original program behavior. Interestingly, the combination of these transformers provides a systematic strategy for the characterization of obfuscating transformers parametric on the program properties to conceal and to reveal. More precisely, the proposed strategy, given a program and the specification of the properties to protect and preserve, provides a characterization of the maximal obfuscation *strategy* for that program, namely of the pool of *all* the computations (execution traces) that could be added to the program in order to obtain the desired semantic obfuscation. Hence, maximal here means that, any other computation (that we could add to the original program semantics) would change also the program semantics property that we aim at preserving. For instance, in the previous example, the I/O semantics of P, that we want to preserve, is $\langle x, s\rangle \rightarrow \langle \top, 6\rangle$ (the only output is the variable s), while we aim at concealing the parity property of s. The intuition beyond the strategy is that, the abstract execution on the parity domain of any obfuscated program must be different from the abstract execution of P on the same domain, while preserving the I/O semantics. Formally, the fix-point of the abstract execution of any obfuscated program must be different from $\langle \top, \top\rangle \rightarrow \langle \mathtt{ev}, \top\rangle \rightarrow \cdots \rightarrow \langle \mathtt{ev}, \mathtt{ev}\rangle$, while its I/O must be $\langle x, s\rangle \rightarrow \langle \top, 6\rangle$. Indeed, in the example, the program $\mathcal{O}(P)$ precisely satisfies this requirement, since it has the same I/O semantics of P, while the abstract execution is $\langle \top, \top\rangle \rightarrow \langle \mathtt{ev}, \top\rangle \rightarrow \cdots \rightarrow \langle \mathtt{ev}, \top\rangle$. Hence, any program transformation, whose semantics is obtained by adding some of the computation given by the strategy to the original program semantics, is an obfuscation method that protects and preserves precisely the considered semantic program properties.

Next, we discuss how the proposed characterization of maximal obfuscation strategy can be used for driving the design of property-based obfuscators. To this end, we consider a recent obfuscating technique [21], where obfuscators are implemented as abstract distorted interpreters. The idea of this obfuscating technique is that, whenever the property to reveal is precisely the I/O semantics, then we can use a suitable distorted interpreter, that by construction, must preserve the I/O semantics, and which is built in order to transform the syntax in a way that a specific property results concealed in the transformed code. It turns out that, in our framework, these distorted interpreters are precisely a way of adding, to the original program, some of the computations contained in the maximal obfuscation strategy for the program.

We validate our results by showing how common obfuscating transformations can be interpreted in the proposed formal framework. For example, we consider the obfuscation technique known as opaque predicate insertion, which adds branches to the control flow of the program that are never executed at run-time. This means that the control flow graph of the obfuscated program corresponds to the control flow graph of the original program, augmented with some spurious paths. In this case, as expected, we are able to show that the computations added by opaque predicate insertion belong to the pool characterized by the maximal obfuscation strategy concealing the original program control flow, and preserving the fact that the original control flow is a sub-graph of the obfuscated program control flow. All the examples proposed show that the obfuscating transformations add computations that obfuscate the property to protect. Indeed, the interpretation of existing code obfuscations in terms of behavioral properties to reveal and conceal allows us to deeply understand the obfuscating behavior of such transformations, namely the relation between what is protected/concealed and what is preserved/revealed.

To summarize, the contributions of the paper are the following:

- Definition and extension of the semantic framework for code obfuscation developed in [14, 16] in order to cope with the semantic specification of the properties, respectively, to protect and preserve (Sect. 3);
- Definition, study and characterization of two function transformers: Revelation, which minimally (in the point-wise order) modifies a function in order to preserve a certain property of the input (Sect. 4), and concealment, which maximally modifies a function in order to hide a certain property of the input (Sect. 5).
- Development of a general framework based on revelation and concealment transformers for characterizing the obfuscation strategy of a particular program with respect to the semantic property to protect and to preserve (Sect. 6).
- Discussion of the applicability (by means of some examples) of the proposed framework by showing how existing obfuscators based on abstract distorted interpreters are indeed a particular way of instantiating property-driven obfuscation strategies (Sect. 7).

2. Preliminaries

2.1. Mathematical notation

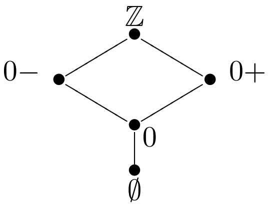
If C and D are sets, then $\wp(C)$ denotes the powerset of C, $C \times D$ denotes the Cartesian product of C and D, $C \smallsetminus D$ denotes the set difference between C and D, $C \subset D$ denotes strict inclusion, and for a function $f : C \to D$, $Y \subseteq C$ and $X \subseteq D$, $f(Y) \stackrel{\text{def}}{=} \{f(y) \mid y \in Y\}$ and $f^{-1}(X) \stackrel{\text{def}}{=} \left\{ x \,\middle|\, f(x) \in X \right\}$. We will often denote $f(\{x\})$ as $f(x)$ and use lambda notation for functions. Function composition $\lambda x.\, f(g(x))$ is denoted $f \circ g$.

A binary relation $\leq \subseteq C \times C$ is a partial order if it is reflexive ($\forall x \in C : x \leq x$), antisymmetric ($\forall x, y \in C : x \leq y \wedge y \leq x \Rightarrow x = y$) and transitive ($\forall x, y, z \in C$). A set C equipped with a partial order relation $\leq$ is called a *poset* and it is denoted with $\langle C, \leq \rangle$. We define the point-wise ordering between functions as follows: given two poset $\langle C, \leq_C \rangle$ and $\langle D, \leq_D \rangle$ and two functions $f, g : C \rightarrow D$, $f \sqsubseteq g$ if and only if $\forall c \in C$ we have that $f(c) \leq_D g(c)$. Let $\langle C, \leq \rangle$ be a poset and let $X \subseteq C$. The least upper bound (lub) of X, denoted $\bigvee X$, is the smallest element $a \in C$ such that $\forall x \in X : x \leq a$. The greatest lower bound (glb) of X, denoted $\bigwedge X$ is the biggest element $a \in C$ such that $\forall x \in X : a \leq x$. We use $x \wedge y$ and $x \vee y$ to denote respectively $\bigwedge\{x, y\}$ and $\bigvee\{x, y\}$. A poset $\langle C, \leq \rangle$ is a *lattice* if $\forall x, y \in C$ we have that $x \vee y \in C$ and $x \wedge y \in C$. A lattice C is a *complete lattice* when each subset of elements has glb and sup, namely when $\forall X \subseteq C : \bigvee C \in C$ and $\bigwedge X \in C$. As usual, $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice C with ordering $\leq$, lub $\vee$, glb $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. Often, $\leq_C$, $\vee_C$, $\wedge_C$, $\top_C$ and $\bot_C$ will be used to denote the underlying ordering, basic operations and elements of the complete lattice C. Consider a poset $\langle C, \leq_C \rangle$with $\bot$. We say that $a \in C$, $a \neq \bot$ is an *atom* of C if $\forall x \in C$ with $x \neq \bot$ we have that $\bot \leq_C x \leq_C a$ implies $x = a$. The notion of *co-atom* is dually defined. A poset C is a *direct set* if each non-empty finite subset of C has lub in C. Given a complete lattice$\langle C, \leq, \vee, \wedge, \top, \bot \rangle$, we define the identity and top functions on C as follows: *id* $\overset{\text{def}}{=} \lambda x.\ x$ and *top* $\overset{\text{def}}{=} \lambda x.\ \top$. Let C and A be complete lattices, function $f : C \rightarrow A$ is *(completely) additive* if f preserves lub of all subsets of C (the empty set included), namely if $\forall X \subseteq C$ we have that $f(\bigvee_C X) = \bigvee_A (f(X))$. Function f is *continuous* when it is monotone and it preserves lubs's of direct sets, namely when for every direct set $X \subseteq C$ we have that $f(\bigvee_C X) = \bigvee_A f(X)$. Co-additivity and co-continuity are dually defined. Let C be a poset with $\bot$ and $\top$. Given an element $x \in C$ we say that $y \in C$ is the complement of x if $x \wedge y = \bot$ and $x \vee y = \top$. A lattice is *complemented* if each one of its elements has complement in the lattice. A lattice is *distributive* if the glb distributes on the lub, i.e., $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$. A complete Boolean algebra is a complete, complemented and distributive lattice, [28]. Given an operator f on a poset $\langle C, \leq_C \rangle$ we say that $x \in C$ is a fix-point of f if $f(x) = x$. The least fix-point of f on $\langle C, \leq_C \rangle$ starting from the lattice element s, when it exists, is denoted by $\mathit{lfp}_s^{\leq_C} f = \bigvee_{n \in \mathbb{N}} f^n(s)$.

2.2. Abstract Interpretation

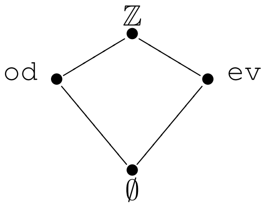
Abstract interpretation is a general theory for specifying and designing approximate semantics of programming languages [10]. Approximation can be equivalently formulated either in terms of Galois connections or closure operators [11]. An *upper closure operator* (uco for short) on a poset $\langle C, \leq_C \rangle$ is an operator $\varphi : C \rightarrow C$ that is monotone ($\forall x \in C : x \leq_C \varphi(x)$), idempotent ($\forall x \in C : \varphi(\varphi(x)) = \varphi(x)$), and extensive ($\forall x \in C : x \leq_C \varphi(x)$). By duality, a *lower closure operator* (lco for short) on a poset $\langle C, \leq_C \rangle$ is an operator $\varphi : C \rightarrow C$ that is monotone, idempotent, and reductive ($\forall x \in C : x \geq_C \varphi(x)$). Let $uco(C)$ and $lco(C)$ denote respectively the set of upper and lower operators on C. Consider a complete lattice C where elements are ordered according to their relative precision. An approximation of the elements of C can be defined as an operator $\varphi \in uco(C)$ that maps each element of $x \in C$ in a less precise element $\varphi(x) \in C$ that represents its approximation in C (recall that $x \leq \varphi(x)$ by definition of upper closure operator). Consider for example the complete lattice given by the power set of integers ordered according to set inclusion: $\langle \wp(\mathbb{Z}), \subseteq \rangle$. The operator *Sign* $: \wp(\mathbb{Z}) \rightarrow \wp(\mathbb{Z})$, operating on the power set of integers, associates each set of integers with its sign: In the following picture we write $0-$ for $\{x \in \mathbb{Z} \mid x \leq 0\}$, 0 for $\{0\}$ and $0+$ for $\{x \in \mathbb{Z}\ \ x \geq 0\}$.

$$\mathit{Sign}(X) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } X = \emptyset \\ \{x \in \mathbb{Z} \mid x \leq 0\} & \text{if } \forall x \in X : x \leq 0 \\ \{x \in \mathbb{Z} \mid x \geq 0\} & \text{if } \forall x \in X : x \geq 0 \\ \{0\} & \text{if } X = \{0\} \\ \mathbb{Z} & \text{otherwise} \end{cases}$$



Analogously, the operator *Par* : $\wp(\mathbb{Z}) \rightarrow \wp(\mathbb{Z})$ associates each set of integers with its parity. In this case, in the picture, we write $\mathtt{ev}$ for $\{x \in \mathbb{Z} \mid x \text{ is even}\}$ and $\mathtt{od}$ for $\{x \in \mathbb{Z} \mid x \text{ is odd}\}$.

$$\mathit{Par}(X) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } X = \emptyset \\ \{x \in \mathbb{Z} \mid z \text{ is even}\} & \text{if } \forall x \in X : x \text{ is even} \\ \{x \in \mathbb{Z} \mid z \text{ is odd}\} & \text{if } \forall x \in X : x \text{ is odd} \\ \mathbb{Z} & \text{otherwise} \end{cases}$$



From these examples,we can clearly see that the abstract elements, in general, correspond to the set of values satisfying the property they represent.

Let us recall the notion of *Moore family* of a complete lattice C: $X \subseteq C$ is a Moore family of C if $X = \{\wedge S \mid S \subseteq X\}$, where $\wedge\varnothing = \top \in X$. Observe that every closure operator $\varphi \in \mathit{uco}(C)$ where C is a complete lattice is uniquely determined by the set of its fix-points $\varphi(C)$ and that this set is a Moore family of C. For instance $\mathit{Par}(\mathbb{Z}) = \{\mathbb{Z}, \mathtt{ev}, \mathtt{od}, \varnothing\}$. Indeed, for a complete lattice C it holds that: if $\varphi \in \mathit{uco}(C)$ then $\varphi(C) \subseteq C$ is a Moore family of C, and for every Moore family $X \subseteq C$ we can define an operator $\varphi_{\mathrm{X}} : C \rightarrow C$ as $\varphi_{\mathrm{X}}(y) \overset{\text{def}}{=} \bigwedge\{x \in X \mid y \leq y\}$, and it is such that $\varphi_{\mathrm{X}} \in \mathit{uco}(C)$ and $\varphi_{\mathrm{X}}(C) = X$. Given a closure $\varphi \in \mathit{uco}(C)$ we say that φ is *meet-uniform on* x if and only if $\varphi(\wedge \left\{ \, y \, \middle| \, \varphi(x) = \varphi(y) \, \right\}) = \varphi(x)$, φ is *meet-uniform* if $\forall x \in C$ we have φ meet-uniform on x [26]. The notion of *join-uniformity* is dually defined.

If $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ is a complete lattice then $\mathit{uco}(C)$ ordered point-wise is also a complete lattice: $\langle \mathit{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\, \top, \mathit{id} \rangle$ where for every $\varphi, \eta \in \mathit{uco}(C)$, $\{\varphi_i\}_{i \in I} \subseteq \mathit{uco}(C)$ and $x \in C$: $\varphi \sqsubseteq \eta$ if and only if $\eta(C) \subseteq \varphi(C)$; $(\sqcap_{i \in I} \varphi_i)(x) = \wedge_{i \in I} \rho_i(x)$; and $(\sqcup_{i \in I} \varphi_i)(x) = x \;\Leftrightarrow\; \forall i \in I.\ \varphi_i(x) = x$, and $\mathit{id} \overset{\text{def}}{=} \lambda x.\, x$. The complete lattice $\langle \mathit{uco}(C), \sqsubseteq \rangle$ of upper closures of C is called the *lattice of abstract interpretations of* C [11]. As argued above, each $\varphi \in \mathit{uco}(C)$ defines a possible abstraction of C where each element $x \in C$ is approximated by the best (more precise) element of $\varphi(C)$ that represents it. Hence, the complete lattice $\langle \mathit{uco}(C), \sqsubseteq \rangle$ is precisely the domain of possible abstractions of C ordered according to their degree of precision. In the following, we will find particularly convenient to identify closure operators (and therefore abstract domains) with their sets of fix-points that represent the property of concrete elements that the closure is able to express.

2.3. Programming Language: Syntax, Semantics and control flow graphs

Let us partially follow the notation used in [21]. We use a toy imperative language for introducing the semantics on which we define our framework, and that we will use in some of the examples. Note that, the choice of the language is not important since our framework is semantic, and therefore language, independent. For abstract interpretation, we need a fine-grained small-step semantics containing program points or similar syntactic information to which abstract values can be bound. Consider a simple imper-

$$\langle \sigma, \textbf{skip}\rangle \to \sigma \qquad \frac{[\![e]\!](\sigma) = n \in \mathbb{V}}{\langle \sigma, x := e\rangle \to \sigma[x \mapsto n]} \qquad \frac{\langle \sigma, \mathtt{C}\rangle \to \sigma'}{\langle \sigma, \mathtt{C};\mathtt{C}_1\rangle \to \langle \sigma', \mathtt{C}_1\rangle} \qquad \frac{\langle \sigma, \mathtt{C}\rangle \to \langle \sigma', \mathtt{C}'\rangle}{\langle \sigma, \mathtt{C};\mathtt{C}_1\rangle \to \langle \sigma', \mathtt{C}';\mathtt{C}_1\rangle}$$

$$\frac{[\![e]\!]\sigma = \mathit{true}}{\langle \sigma, \textbf{if } e \textbf{ then } \mathtt{C}_0 \textbf{ else } \mathtt{C}_1 \textbf{ fi}\rangle \to \langle \sigma, \mathtt{C}_0\rangle} \qquad \frac{[\![e]\!]\sigma = \mathit{false}}{\langle \sigma, \textbf{if } e \textbf{ then } \mathtt{C}_0 \textbf{ else } \mathtt{C}_1 \textbf{ fi}\rangle \to \langle \sigma, \mathtt{C}_1\rangle}$$

$$\frac{[\![e]\!]\sigma = \mathit{true}}{\langle \sigma, \textbf{while } e \textbf{ do } \mathtt{C} \textbf{ endw}\rangle \to \langle \sigma, \mathtt{C}; \textbf{while } e \textbf{ do } \mathtt{C} \textbf{ endw}\rangle} \qquad \frac{[\![e]\!]\sigma = \mathit{false}}{\langle \sigma, \textbf{while } e \textbf{ do } \mathtt{C} \textbf{ endw}\rangle \to \sigma}$$

Table 1
Small-step semantics of $\mathfrak{L}$

ative language $\mathfrak{L}$ parameterized by an expression sub-language e that contains the usual arithmetic and boolean expressions and functions:

$$\begin{aligned}&\mathtt{P} ::= \textbf{input } x;\ \mathtt{C};\ \textbf{output } x;\\&\mathtt{C} ::= \textbf{skip} \mid x := e \mid \mathtt{C};\mathtt{C} \mid \textbf{while } e \textbf{ do } \mathtt{C} \textbf{ endw} \mid \textbf{if } e \textbf{ then } \mathtt{C} \textbf{ else } \mathtt{C} \textbf{ fi}\end{aligned}$$

The single commands of the language $\mathfrak{L}$ are in the following denoted with $\mathtt{c}$. In Table 1 we omit the formal semantics of **input** x which corresponds to a dynamic assignment to x, and of **output** x which corresponds to the *visualization* of the output result. We consider a quite standard operational semantics of the language. Let $\mathit{Prog}_{\mathfrak{L}}$ be a set of programs in the language $\mathfrak{L}$ (in the following simply denoted *Prog* when the language can be left implicit), $\mathit{Var}(\mathtt{P})$ the set of all the variables in $\mathtt{P}$, and $\mathbb{PL}_{\mathtt{P}}$ be a set of program lines of $\mathtt{P} \in \mathit{Prog}_{\mathfrak{L}}$ containing a special notation ϵ for the empty program line, $\mathbb{V}$ be the set of values, and $\mathtt{M} \overset{\text{def}}{=} \mathit{Var}(\mathtt{P}) \longrightarrow \mathbb{V}$ be a set of possible program memories. When a command $\mathtt{c}$ belongs to a program $\mathtt{P}$ we write $\mathtt{c} \in \mathtt{P}$, and we define the auxiliary functions $\mathtt{Stm}_{\mathtt{P}} : \mathbb{PL}_{\mathtt{P}} \to \mathit{Prog}_{\mathfrak{L}}$ be such that $\mathtt{Stm}_{\mathtt{P}}(l) = \mathtt{c}$ if $\mathtt{c}$ is the command in $\mathtt{P}$ at program line l (denoted ${}^{l}\mathtt{c}$) and $\mathtt{Pc}_{\mathtt{P}} = \mathtt{Stm}_{\mathtt{P}}{}^{-1} : \mathit{Prog}_{\mathfrak{L}} \to \mathbb{PL}_{\mathtt{P}}$ such that $\mathtt{Pc}_{\mathtt{P}}(\mathtt{c}) = l$ if command $\mathtt{c} \in \mathtt{P}$ is stored at location l, with the simple extension to sequences of instructions $\mathtt{Pc}_{\mathtt{P}}(\mathtt{c};\mathtt{C}) = \mathtt{Pc}_{\mathtt{P}}(\mathtt{c})$. Then, letting $\sigma \in \mathtt{M}$, we define the semantics of $\mathfrak{L}$ in Table 1, where x are variables, e are (potentially boolean) expressions, $[\![\cdot]\!]$ is the evaluation of expressions, and where we write $\langle \sigma, \mathtt{C}\rangle \Downarrow \langle \sigma', \mathtt{C}'\rangle$ for the execution of $\mathtt{C}$ in the memory σ with continuation $\mathtt{C}'$, and $\langle \sigma, \mathtt{C}\rangle \Downarrow \sigma$ for termination in state σ. We can formally characterize the small-step operational semantics of programs. Let $\Sigma = (\mathtt{M} \times \mathit{Prog}_{\mathfrak{L}}) \cup \mathtt{M}$ be the set of (nonterminating) states, containing the actual memory and the code to execute, and $s \in \Sigma\langle \sigma, \mathtt{C}\rangle \in \Sigma$. The function $f_{\mathfrak{L}} : \Sigma \longrightarrow \wp(\Sigma)$ is such that:

$$f_{\mathfrak{L}}(s) = \begin{cases} \left\{\, \langle \sigma', \mathtt{C}'\rangle \,\middle|\, \langle \sigma, \mathtt{C}\rangle \Downarrow \langle \sigma', \mathtt{C}'\rangle \,\right\} \cup \left\{\, \sigma' \,\middle|\, \langle \sigma, \mathtt{C}\rangle \Downarrow \sigma' \,\right\} & \text{if } s = \langle \sigma, \mathtt{C}\rangle \\ s & \text{if } s = \sigma \in \mathtt{M} \end{cases}$$

It is worth noting that for deterministic languages like $\mathfrak{L}$, this set contains only one state or one memory.

We use the transfer function to define the small-step operational semantics, denoted by $[\![\cdot]\!]$, as the set of traces of terminating and infinite executions. Let Σ^* denote the set of all the finite traces on Σ, and Σ^ω the set of all infinite traces, we denote by $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ the set of all possible traces. Let $\mathtt{P} \in \mathit{Prog}$ and

$s \in \Sigma$, $[\![\mathtt{P}]\!] : \Sigma \longrightarrow \wp(\Sigma^{\infty})$

$$[\![\mathtt{P}]\!](s) \overset{\text{def}}{=} \left\{ \tau \in \Sigma^* \;\middle|\; \begin{array}{l} \forall 1 \leq i < |\tau|.\, \tau_i \in f_{\mathfrak{L}}(\tau_{i-1}), \\ \tau_{\vdash} = s,\ \tau_{\dashv} \in \mathtt{M} \end{array} \right\} \cup \left\{ \tau \in \Sigma^{\omega} \;\middle|\; \begin{array}{l} \forall 1 \leq i.\, \tau_i \in f_{\mathfrak{L}}(\tau_{i-1}), \\ \tau_{\vdash} = s \end{array} \right\}$$

where τ_i denotes the i-th state in the trace τ, $|\tau|$ denotes the length of trace τ, $\tau_{\vdash} \overset{\text{def}}{=} \tau_0$, $\tau_{\dashv} \overset{\text{def}}{=} \tau_{|\tau|-1}$. See [9] for details on the fix-point construction of operational semantics. In [9] several kind of semantics are defined as abstractions of small-step semantics, let us recall denotational (I/O) semantics $[\![\mathtt{P}]\!]^{\mathfrak{I}} : \Sigma \longrightarrow \wp(\Sigma \cup \{\bot\})$:

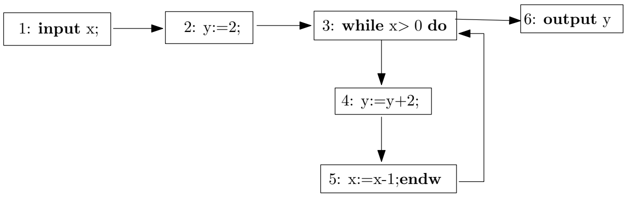
$$[\![\mathtt{P}]\!]^{\mathfrak{I}}(s) \overset{\text{def}}{=} \{\, s' \,|\, \exists \tau \in [\![\mathtt{P}]\!](s) \cap \Sigma^*,\ \tau_{\dashv} = s' \,\} \cup \{\, \bot \,|\, \exists \tau \in [\![\mathtt{P}]\!](s) \cap \Sigma^{\omega} \,\}$$

In the following, we abuse notation by denoting with $[\![\cdot]\!]$ and $[\![\cdot]\!]^{\mathfrak{I}}$ also the corresponding additive lift to set of states, i.e., given $S \subseteq \sigma$ we have that $[\![\mathtt{P}]\!](S) = \bigcup\{[\![\mathtt{P}]\!](s) \mid s \in \Sigma\}$ and $[\![\mathtt{P}]\!]^{\mathfrak{I}}(S) = \bigcup\{[\![\mathtt{P}]\!]^{\mathfrak{I}}(s) \mid s \in \Sigma\}$.

Programs are often analyzed by means of their control flow graph (CFG). A CFG models the flow of control between program instructions. We consider the CFG as given by a directed graph $\mathtt{G} = \langle N, E \rangle$, where each node $n \in N$ corresponds to an instruction, and each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from instruction n_i to instruction n_j [8]. Note that, the execution of a CFG corresponds to the set of state traces due to the execution of the commands on the considered CFG.

Example 1 *Consider the following program and corresponding single-command CFG [24]*

```
1.input x;
2.y := 2;
3.while x > 0 do
     4.y := y + 2;
     5.x := x − 1
  endw
6.output y;
```



In this case, since the program is deterministic, we have only one execution of the CFG for each input. For instance, the execution (denoted by the sequence of executed program points) starting from the state where $x = 1$ *is:* $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$.

2.4. Code Obfuscation

In this section, we recall the standard definition of code obfuscation. Code obfuscation was first defined by Collberg et al. [7] as follows.

Definition 1 ([7]) *A syntactic program transformation* $\mathbb{T}$: *Prog* $\longrightarrow$ *Prog is an obfuscator if:*

- *the transformation* $\mathbb{T}$: *Prog* $\longrightarrow$ *Prog is potent, namely it makes programs more difficult to analyse;*
- P *and* $\mathbb{T}(\mathtt{P})$ *have the same observational behavior, i.e., if* P *fails to terminate or it terminates with an error condition then* $\mathbb{T}(\mathtt{P})$ *may or may not terminate; otherwise* $\mathbb{T}(\mathtt{P})$ *must terminate and produce the same result as* P.

The second point 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 only the finite behaviors of the original and of the transformed program. This definition of code obfuscation makes it clear that the aim of code obfuscation is to obstruct program analysis, by making it more complex, thus concealing some information while preserving the observational behavior of programs (i.e., program denotational semantics). For instance, consider the following slight abstraction of the I/O semantics defined in Section 2.3: Let us consider $[\![\mathtt{P}]\!]^{\mathcal{I}}_{\mathcal{E}}(s) : \Sigma \rightarrow \wp(\Sigma \cup \{\bot\})$ with $\mathcal{E}$ denoting the set of states denoting error conditions:

$$[\![\mathtt{P}]\!]^{\mathcal{I}}_{\mathcal{E}}(s) \stackrel{\text{def}}{=} \left\{ s' \,\middle|\, s' \in [\![\mathtt{P}]\!]^{\mathcal{I}}(s),\ s' \notin \mathcal{E} \right\} \cup \left\{ \bot \,\middle|\, (\exists s' \in [\![\mathtt{P}]\!]^{\mathcal{I}}(s),\ s' \in \mathcal{E}) \ \vee\ \bot \in [\![\mathtt{P}]\!]^{\mathcal{I}}(s) \right\}$$

Namely, we precisely observe the final output only of terminating traces which do not terminate with an error condition. The definition of obfuscation of Collberg et al. requires that $[\![\mathtt{P}]\!]^{\mathcal{I}}_{\mathcal{E}} = [\![\mathbb{T}(\mathtt{P})]\!]^{\mathcal{I}}_{\mathcal{E}}$. In other words Definition 1 says that obfuscation has to *reveal* $[\![\cdot]\!]^{\mathcal{I}}_{\mathcal{E}}$, which is an abstraction (i.e., a property) of concrete semantics $[\![\cdot]\!]$ [9].

A typical example of code obfuscation is the insertion of fake branches through opaque predicates [7]. A true (resp. false) opaque predicate is a predicate that always evaluates to *true* (resp. *false*). Program functionality is preserved by inserting the intended behavior in the always taken branch and buggy code in the never executed branch. This clearly confuses a static attacker that is not aware of the constant value of the opaque predicate and has to analyze both branches as possible program executions. In both cases the constant value of the predicate has to be difficult to deduce for an external observer that sees both branches as possible. However, the notions of transformation potency (or program complexity), of observational behavior and of attacker on which this definition relies are still informal. It is clear that in order to formally and deeply understand the potentiality and limits of code obfuscation it is necessary to formally specify the above mentioned notions.

In [14, 16] the informal definition of code obfuscation of Collberg et al. has been generalized and placed in a theoretical framework based on program semantics and abstract interpretation. The idea is to introduce a formal model of malicious host attacks and of code transformations that allows a rigorous specification of the amount of "obscurity" added by a transformation to program semantics in the abstract interpretation framework.

3. The semantic framework for code obfuscation

When speaking of code obfuscation, we refer to program transformations that aim at obstructing reverse engineering by making it more difficult for attackers to obtain precise information regarding the original program. In general, we can say that any obfuscation technique is build for preserving some *program features* (in particular the I/O functionality) and for deceiving some other program features, namely for deceiving any feature that an attacker may exploit to perform reverse engineering. Let us explain our idea by means of examples.

An example, data-type obfuscation, has been given in the introduction, where the deceived feature is a semantic property, the parity of the computed values, that can be analyzed by static analysis, and in particular which is modeled by abstract interpretation [10, 11]. But, what we are proposing here is not limited only to this kind of program features, since data-type obfuscation is not the only obfuscation technique where we can semantically characterize what is revealed and what is concealed. Consider,

```
original(){
    1. int c, nl = 0, nw = 0, nc = 0, in;
    2. in = false;
    3. while((c = getchar()) != EOF){
        4. nc++;
        5. if(c == ' ' || c =='\n' || c == '\t') in = false;
        6. else if(in == false) {in=true; nw++;}
        7. if(c == '\n')nl++;
        }
8. out(nl,nw,nc); }
```

	c	nc	nw	nl
c				
nc	√	√		
nw	√		√	
nl	√			√

```
obfuscated(){
    1. int c, nl = 0, nw = 0, nc = 0, in;
    2. in = false;
    3. while((c = getchar()) != EOF){
        4. nc++;
        5. if(c == ' ' || c =='\n' || c == '\t') in = false;
        6. else if(in == false) {in=true; nw++;}
        7. if(c == '\n'){if(nw <= nc)nl++};
        8. if(nl > nc) nw = nc+nl;
        9. else if(nw > nc) nc = nw - nl;
        }
10. out(nl,nw,nc); }
```

	c	nc	nw	nl
c				
nc	√	√	√	√
nw	√	√	√	√
nl	√	√	√	√

Fig. 1. Original and obfuscated programs of Example 2.

for instance, slicing obfuscation where we suppose that the attacker performs reverse engineering by extracting slices of the program. Hence, since standard algorithms for program slicing [29] are based on the computation of data dependencies, slicing obfuscation aims at deceiving precisely data dependencies, by adding *fake* dependencies, i.e., syntactic dependencies which do not corresponds to semantic ones [25]. In this way, any slicing algorithm based on data dependencies will compute imprecise slices.

Example 2 *Consider the word count program in Fig.*1 *[35]. Note that the example is taken verbatim from [35] and it is written in C language*[1]*. It takes in a block of text and outputs the number of lines (nl), words (nw) and characters (nc). The syntactic program transformation* $\mathbb{T}$ *modifies line* 7 *by adding a true opaque predicate and adds lines* 8 *and* 9 *with false opaque predicates [35], i.e.,* $\mathbb{T}(\text{original}) = \text{obfuscated}$. *Let us consider both explicit and implicit data dependencies of the two programs. We say that there is a data dependence of variable* v_1 *from variable* v_2*, denoted* (v_1, v_2)*, when* v_1 *is defined in terms of* v_2 *or when there is a definition of* v_1 *in a branch of the program that is controlled by a condition in which appears* v_2*. The table on the right of each program reports the dependencies among variables in the program. Suppose the slicing criterion looks for* nl *at the end of the program, then the slice of the original program is on the left in Fig.* 2*, while the slice of the obfuscated program is on the right. Hence, the above transformation conceals the real data dependencies of the program by adding fake*

[1] We decided not to translate this code in our toy imperative language for readability, for instance our language does not have an equivalent for *getchar* returning the next char in a file, or **out** as output instruction. However, this is not a problem since our framework is language independent.

```
or-slice-nl(){
    1. int c, nl = 0;
    3. while ((c = getchar()) != EOF){
        7. if(c == '\n')nl++;
        }
8. out(nl); }
```

```
obf-slice-nl(){
    1. int c, nl = 0, nw = 0, nc = 0, in;
    2. in = false;
    3. while (( c = getchar()) != EOF){
        4. nc++;
        5. if(c == ' ' || c =='\n' || c == '\t') in = false;
        6. elseif(in == false) {in=true; nw++;}
        7. if(c == '\n'){if(nw <= nc)nl++};
        }
10. out(nl); }
```

Fig. 2. Slices of the programs of Example 2.

dependencies between program variables, hence the obfuscation, while preserving the I/O semantics, has concealed the slicing computation on $\mathtt{P}$ *with respect to* nl*, due to the fake dependencies added by* $\mathbb{T}$.

3.1. Semantic code obfuscation.

In this section, we describe the semantics-based approach to code obfuscation [14, 16, 21] in terms of the properties, respectively, to protect and to preserve.

Attacker model. In this context, the typical attacker aims at performing reverse engineering of programs for stealing or copying ideas. Reverse engineering attempts to understand the inner workings of programs by performing some kind of analysis. Reverse-engineering techniques include both static program analysis (e.g., data flow analysis, control flow analysis, alias analysis, program slicing) and dynamic program analysis which may also interact with the program execution (e.g., dynamic testing, profiling, program tracing). Moreover, reverse engineering techniques may be performed either automatically or manually, and may take advantage from the possibility of attacking a program several times on different environments. Due to the complexity and the diversity of the techniques that an attacker may perform, it is clear that when providing a formal framework we have necessarily to make some assumptions, obviously admitting some limitation in the model. In the particular context of the framework that we propose in this paper, we fix the model of the attacker at the semantic level, in order to allow a sufficient degree of generality, even if in this way we are not able to capture any possible attack. For instance, we cannot model manual attacks interacting with the execution. Nevertheless, we model any attack that extracts features through a specific observation of the program execution, i.e., of the program operational semantics (see Sect. 2.3).

In the following, we follow the ideas presented in [14, 16, 21, 25]. Given a feature of interest, we model the corresponding semantic property as a function φ mapping any set of executions X to the sets of all the executions having the same feature of interest as the executions in X. This implies that φ is extensive (i.e., $\varphi(X) \supseteq X$), namely it approximates by adding *noise*, it is idempotent since the whole approximation is added in one shot, and finally it is monotone, preserving the approximation order. Namely it is an upper closure operator and the framework beneath is abstract interpretation [10, 11]. As we have seen in the previous section, the set $\wp(\Sigma^\infty)$ can be used to represent the set of possible program semantics, where Σ denotes the set of possible program states. Thus, we view attackers as the properties of program behaviors that they can observe/analyze, i.e., as $uco(\wp(\Sigma^\infty))$ over the domain of program operational semantics. In other words, the attacker is the map (uco) associating with each semantics S (set of executions) the smallest set of executions containing S and satisfying a corresponding invariant (the S observable property).

Hence, in the following of this paper, we only focus on program features, both preserved/revealed and protected/concealed, that are *properties* of the operational semantics, where by property we *extensionally* mean the set of all the executions having the desired particular feature. Hence, for instance, if we observe the parity of values, then a trace $\langle\{x \mapsto 2\}, \mathtt{C}_1\rangle \to \langle\{x \mapsto 3\}, \mathtt{C}_2\rangle$ is approximated in the set of traces where only the parity of values is observable, i.e., $\{\, \{x \mapsto ev\} \to \{x \mapsto od\} \mid ev \in 2\mathbb{Z}, od \in 2\mathbb{Z}+1 \,\}$. Analogously, if we observe data dependencies, the property of a trace is the set of all the computations inducing the same set of dependencies, that in this case can be identified simply with the information observed, i.e., the set of data dependencies. More formally, we can define the data dependency approximation $\mathcal{D} \in uco(\wp(\Sigma^\infty))$ which approximates any set of computations in the set of all the data dependencies among variables generated during the computation (e.g, by means of program dependence graphs), e.g., it approximates the operational semantics of programs in Fig. 1 in the set of dependencies represented in the tables on their right. In this case, we observe that $\mathcal{D}(\llbracket \mathit{original} \rrbracket) \neq \mathcal{D}(\llbracket \mathit{obfuscated} \rrbracket)$, meaning that an external observer is not able to derive, from the analysis of the obfuscated program, the precise data dependencies holding in the original program, overestimating this information.

Note that, in this framework, the fact that we model attackers as the set of executions, or as an approximation of these executions, that they can observe, completely ignores the way in which the attacker performs the observation. In other words, what we propose depends on *what* the attacker observes, while it ignores *how* the attacker collects the observations of programs.

Obfuscation model. As argued above, an obfuscation is a program transformation that aims to confuse some properties of programs while preserving program behavior to some extent. This means that there are properties of the program behavior that are preserved by the obfuscation (typically the I/O semantics) and properties that are not preserved (the ones that the obfuscation aims at confusing). In [21], the authors specify when a program transformation is an obfuscator, in the semantic setting. Following this view, we obtain the following characterisation of an obfuscator:

- An obfuscation transformer $\mathfrak{O}$ has to preserve at least the I/O functionality of programs, namely the I/O semantics of a program $\mathtt{P}$ and of its obfuscated version $\mathfrak{O}(\mathtt{P})$ have to be the same. More formally, let $\mathcal{I} \in uco(\wp(\Sigma^\infty))$ be the observation of the only I/O relation of the operational semantics of a program [9]

  $$\mathcal{I}(X) = \left\{\, \tau \in \Sigma^* \,\middle|\, \exists \tau' \in X \cap \Sigma^*.\ \tau_\dashv = \tau'_\dashv,\ \tau_\vdash = \tau'_\vdash \,\right\} \cup \left\{\, \tau \in \Sigma^\omega \,\middle|\, \exists \tau' \in X \cap \Sigma^\omega,\ \tau_\vdash = \tau'_\vdash \,\right\}$$

  This is the closure operator corresponding to the semantics $\llbracket\cdot\rrbracket^{\mathcal{I}}$ defined in Section 2.3. Then $\mathcal{I}(\llbracket \mathtt{P} \rrbracket) = \mathcal{I}(\llbracket \mathfrak{O}(\mathtt{P}) \rrbracket)$.
- An obfuscator has to add confusion with respect to some properties that are revealed by the non-obfuscated program $\mathtt{P}$, thus generating an obfuscated program $\mathfrak{O}(\mathtt{P})$ from which the confused properties cannot be precisely extracted. Namely, $\varphi(\llbracket \mathtt{P} \rrbracket) \neq \varphi(\llbracket \mathfrak{O}(\mathtt{P}) \rrbracket)$. In this case, we can say that the obfuscator is φ-potent for $\mathtt{P}$ [14, 16].

In the present framework, we say that the obfuscator preserves/reveals $\mathcal{I}$, while it conceals/protects φ, since φ cannot be extracted from the obfuscated program as it was on the original one. It is worth noting that in this way any property more abstract than $\mathcal{I}$ is automatically preserved.

In the slicing obfuscation example we can prove that $\mathcal{I}(\llbracket \mathit{original} \rrbracket) = \mathcal{I}(\llbracket \mathit{obfuscated} \rrbracket)$. Namely the I/O semantics of the original program can be precisely derived by the observation of its obfuscated version. Hence, the obfuscation $\mathbb{T}$ conceals $\mathcal{D}$, while it preserves/reveals $\mathcal{I}$.

Note that this semantic approach to obfuscation allows us to characterize the obfuscating behavior of an obfuscation in terms of the most concrete property it preserves of program behavior, namely of program semantics. Let us denote with $\delta_{\mathbb{T}} \in uco(\wp(\Sigma^\infty))$ the most concrete property preserved by obfuscation $\mathbb{T} : \mathit{Prog} \to \mathit{Prog}$ on the semantics of programs [16]. The idea is that the most concrete semantic property $\delta_{\mathbb{T}} \in uco(\wp(\Sigma^\infty))$ can be used to characterize the semantic properties, namely the attackers, that are obstructed by the obfuscation $\mathbb{T}$ and the ones that are not. Indeed, any attacker that is modeled as a property $\varphi \in uco(\wp(\Sigma^\infty))$ implied by the most concrete preserved property, i.e., $\delta_{\mathbb{T}} \sqsubseteq \varphi$, is not obstructed by obfuscation $\mathbb{T}$; while any attacker interested in a property φ not implied by $\delta_{\mathbb{T}}$, i.e., $\delta_{\mathbb{T}} \not\sqsubseteq \varphi$, is defeated. Moreover, the mapping of syntactic program transformations to the lattice of abstract interpretations allows us to measure, reason and compare the potency and efficiency of different obfuscating transformations. For example, in [13, 15, 16] the proposed model of attackers and obfuscation has been used for understanding the class of attackers that are confused by opaque predicate insertion. In particular, the cited works consider the insertion of numerical opaque predicates such as $\forall x \in \mathbb{Z} : 2\ \mathit{mod}\ (x^2 + x)$. The attackers defeated by the opaque predicates are those that do not understand the always true value of these predicates. In fact, these attackers have to consider both the possible branches leading to a loss of precision of the static program analysis. These attackers are modeled by numerical abstract domains that do not allow precisely computing the invariant of the opaque predicate. For this reason, these abstract domains evaluate the predicate to $\top$, modeling that it can be either false or true. In this way, the attackers see both branches as possible. Consider, for example, an attacker interested in the Sign property, namely an attacker modeled by the abstract domain $\mathit{Sign} = \{\emptyset, 0-, 0, 0+, \mathbb{Z}\}$. This attacker would not be able to capture the opacity of the above mentioned opaque predicate $\forall x \in \mathbb{Z} : 2\ \mathit{mod}\ (x^2 + x)$. Indeed, consider $x \in \mathbb{Z}$ such that $x < 0$. Computing the truth value of this opaque predicate for $x < 0$ on *Sign* means to compute $2\,\mathbf{mod}\,((0-)^{\mathbf{2}}\ \mathbf{+}\ 0-)$, where **mod**, $^{\mathbf{2}}$ and **+** are the approximation of the module, square and addition operators on the domain of signs. Thus, $2\,\mathbf{mod}\,((0-)^{\mathbf{2}}\ \mathbf{+}\ 0-) = 2\,\mathbf{mod}\,(0+\ \mathbf{+}\ 0-)$ since the square of any integer number returns a positive number. Next, $2\,\mathbf{mod}\,(0+\ \mathbf{+}\ 0-) = 2\,\mathbf{mod}\,\mathbb{Z}$ because the sum of a positive number with a negative number can return either a positive or a negative number. Thus, $2\,\mathbf{mod}\,\mathbb{Z} = \top$ because an integer number may or may not be a multiple of 2. Consider for example the following program $\mathtt{P}$ and its obfuscation $\mathbb{T}(\mathtt{P})$ obtained through the insertion of the considered opaque predicate:

$$\mathtt{P} : \left\{ \begin{array}{l} {}^{1\cdot}\textbf{input}\ x; \\ {}^{2\cdot}x := |x|; \\ {}^{3\cdot}\textbf{output}\ x; \end{array} \right. \qquad\qquad \mathbb{T}(\mathtt{P}) : \left\{ \begin{array}{l} {}^{1\cdot}\textbf{input}\ x; \\ {}^{2\cdot}\textbf{if}\ 2\ \mathrm{mod}\ (x^2 + x) \\ \qquad \textbf{then}\ {}^{3\cdot}x := |x|\ \textbf{else}\ {}^{4\cdot}x := -|x|\ \textbf{fi} \\ {}^{5\cdot}\textbf{output}\ x; \end{array} \right.$$

The attacker modeled by *Sign* precisely infers that the output value of x in program $\mathtt{P}$ is always positive, while it is not able to extract this information from the obfuscated program $\mathbb{T}(\mathtt{P})$. Indeed, in the abstract semantics of $\mathbb{T}(\mathtt{P})$ the attacker *Sign* has to consider both branches as possible and therefore the analysis of the sign of the output on $\mathbb{T}(\mathtt{P})$ returns $\mathbb{Z}$ meaning that the attacker is not able to understand that the program always outputs a positive number. This means that, the property of *Sign* is not preserved on the transformed program $\mathbb{T}(\mathtt{P})$ by the considered obfuscation. In [15] these kind of observations have led to the design of an efficient opaque predicate detector for a class of numerical opaque predicates.

Syntactic vs semantic transformations. While code obfuscation is a syntactic program transformation $\mathbb{T} : \mathit{Prog} \to \mathit{Prog}$ and it transforms a program into another program, we have decided to model the obfuscating behavior of $\mathbb{T}$ on the semantics of programs. This means that we are interested in the effects that obfuscation $\mathbb{T}$ has on the semantics of programs. To this end, we find very useful the formal framework introduced by Cousot and Cousot [12], in which syntactic program transformations are related to their semantic counterpart and vice-versa.

The formal definition of the relation between syntactic and semantic program transformations given by Cousot and Cousot [12] allows us to reason on the effects that code transformations have on semantics. We consider *Prog* to be the domain of programs up to semantic equivalence, where two programs $\mathtt{P}$ and $\mathtt{Q}$ are semantically equivalent if $[\![\mathtt{P}]\!] = [\![\mathtt{Q}]\!]$, namely if they have the same semantics. In [12] programs are seen as abstractions of their semantics and this is formalized in the abstract interpretation framework. In particular, the semantic domain $\langle \wp(\Sigma^\infty), \subseteq \rangle$ is abstracted in the syntactic domain $\langle \mathit{Prog}, \trianglelefteq \rangle$, where $\trianglelefteq$ is the order induced on programs, namely $\mathtt{P} \trianglelefteq \mathtt{Q} \overset{\text{def}}{=} [\![\mathtt{P}]\!] \subseteq [\![\mathtt{Q}]\!]$, the abstraction of $X \in \wp(\Sigma^\infty)$ is the semantics of the simplest program $\mathit{code}([\![X]\!])$ (smallest number of instructions) that upper-approximates X. This means that, it is possible to associate to every syntactic program transformation $\mathbb{T} : \mathit{Prog} \to \mathit{Prog}$ its semantic counterpart $T : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$, such that $T([\![\mathtt{P}]\!]) = [\![\mathbb{T}(\mathtt{P})]\!]$ [12] and vice versa: $T([\![\mathtt{P}]\!]) = [\![\mathbb{T}(\mathit{code}([\![\mathtt{P}]\!]))]\!]$ and $\mathbb{T}(\mathtt{P}) = \mathit{code}(T([\![\mathtt{P}]\!]))$.

Observe that the equation $\mathbb{T}(\mathtt{P}) = \mathit{code}(T([\![\mathtt{P}]\!]))$ expresses a syntactic transformation as an abstraction of the semantic transformation. Interestingly, starting from this formalization it is possible to derive a systematic methodology for designing syntactic transformations from semantic ones [12]. Of course, when the semantic transformation T relies on results of undecidable problems, any effective algorithm $\mathbb{T}$ that tries to implement T would be an approximation of the ideal transformation $\mathit{code} \circ T \circ [\![\cdot]\!]$. Thus, in general $T([\![\mathtt{P}]\!]) \subseteq [\![\mathbb{T}(\mathtt{P})]\!]$, or equivalently $\mathit{code}(T([\![\mathtt{P}]\!])) \trianglelefteq \mathbb{T}(\mathit{code}([\![\mathtt{P}]\!]))$.

In the context of code obfuscation, the formal framework of Cousot and Cousot could be used to:

(1) *model obfuscation potency:* reason on the effects that an obfuscation has on program semantics in order to deeply understand the semantic properties that are protected, i.e., concealed by the obfuscation, with respect to attackers modeled as semantic program properties

(2) *specify property-driven obfuscation:* given a semantic property to protect φ and a semantic property to preserve δ, develop a semantic transformation that conceals φ and reveals δ and uses this semantic characterization as a “measure" of optimality for any syntactic transformation implementing the corresponding semantic code obfuscation.

In other words, it provides a common ground, where syntactic obfuscations can be semantically analysed and compared with respect to the properties they are able to protect. Based on the investigation of point (1), presented in [14, 16], we address here point (2).

From now on, we consider the semantic counterpart of code obfuscation, since it is at the semantic level that we can formally understand what is concealed and what is revealed. Indeed, *studying obfuscation at the semantic level means studying its ideal behavior that will be necessarily approximated during the implementation process*. In the following, an obfuscation of a property $\varphi \in uco(\wp(\Sigma^\infty))$ and a program $\mathtt{P}$ is a semantic program transformation $T : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ concealing φ on the semantics of $\mathtt{P}$, i.e., such that $\varphi([\![\mathtt{P}]\!]) \neq \varphi(T([\![\mathtt{P}]\!]))$. We would say that a semantic program transformation $T : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ is an obfuscation for φ if there exists a program for which T is an obfuscation, namely if $\exists \mathtt{P} \in \mathit{Prog} : \varphi([\![\mathtt{P}]\!]) \neq \varphi(T([\![\mathtt{P}]\!]))$

3.2. The challenge: property-driven obfuscations.

The formal framework described above [14,16] models code obfuscation in terms of the most concrete semantic property preserved by the program transformation and this allows us to compare the potency of different obfuscating transformations and sometimes also their resilience [13, 15], namely how well a transformation holds up under attack from a de-obfuscator [7]. However, this theoretical investigation does not provide any insight in the design of an efficient obfuscation. Indeed, *what is still missing is a general strategy for designing and evaluating obfuscations given the specification of the property φ to protect, i.e., to conceal, and of the property δ to preserve, i.e., to reveal.* This is exactly the high level goal of our investigation. More specifically, we investigate a general framework of function transformers that aim at minimally or maximally transform a function in order to reveal or conceal a given semantic property of a program. To this end, we first model and characterize the minimal transformations that preserve a certain property, later called *revelation*. To preserve means to leave unchanged and, therefore, to *reveal* the property of the original program in the transformed/obfuscated program. Moreover, we model and characterize the maximal transformation that protect a given property, later called *concealment* transformers. To protect a property of the original program means to change the property and, in this way, to *conceal* it. Next, we show how the combination of revelation and concealment can be used for characterizing an obfuscating transformation from the specification of the property φ to be concealed and the property δ to be revealed. What we obtain, in this way, is the characterization of the semantic transformations that exhibit the intended obfuscating behavior. This characterization could then be used to drive the design strategy for syntactic code obfuscations that implement the desired semantic behavior.

4. Modeling Revelation

In this section, we investigate the semantic transformers minimally modifying a program semantics, in order to reveal a given property. We first provide a formal definition of the revelation transformers as a generic function transformer, and then we prove some of their interesting properties. Afterwards, we provide a characterization of the revelation transformers making more explicit how the construction of the revelation is based on the preservation of the property to be revealed. Then, we show how existing program transformations are indeed revelations. To conclude, we discuss how the revelation transformers relate to the semantics-based notion of code obfuscation.

Consider the complete lattice $\langle L \to L, \sqsubseteq, \sqcup, \sqcap, \lambda x.\bot, \lambda x.\top\rangle$ of functions over a complete lattice L, where functions are ordered point-wise. Given a function $f : L \to L$ and a property δ modeled as abstraction of L, i.e., $\delta \in uco(L)$, we define the two function transformers $\mathcal{R}^{\uparrow}_{\delta}$ and $\mathcal{R}^{\downarrow}_{\delta}$ that aim at computing the two functions closest to f in the domain $L \to L$, respectively from above and from below, revealing the property δ. We consider both forms of revelation because there are two possible ways to minimally transform a function in order to make it reveal a property (from above and from below).

Definition 2 (Minimal revelation) *Let L be a complete lattice and $\delta \in uco(L)$. The operators $\mathcal{R}^{\uparrow}_{\delta}, \mathcal{R}^{\downarrow}_{\delta}$: $(L \to L) \to (L \to L)$ are the minimal revelation from above and from below for δ:*

$$\mathcal{R}^{\uparrow}_{\delta}(f) \overset{def}{=} \bigsqcap \left\{\, g : L \to L \,\middle|\, \forall x \in L.\ \delta(x) = \delta(g(x)), f \sqsubseteq g \,\right\}$$
$$\mathcal{R}^{\downarrow}_{\delta}(f) \overset{def}{=} \bigsqcup \left\{\, g : L \to L \,\middle|\, \forall x \in L.\ \delta(x) = \delta(g(x)), g \sqsubseteq f \,\right\}$$

Observe that, it is interesting to study the minimal revelation from above $\mathcal{R}^{\uparrow}_{\delta}(f)$ and from below $\mathcal{R}^{\downarrow}_{\delta}(f)$ for f when: (*) $\mathcal{R}^{\uparrow}_{\delta}(f)$ and $\mathcal{R}^{\downarrow}_{\delta}(f)$ do *not trivially* transform f, i.e., they do not transform f in the top $(\lambda x.\top)$ or the bottom $(\lambda x.x)$ of the functional domain; (**) $\mathcal{R}^{\uparrow}_{\delta}(f)$ and $\mathcal{R}^{\downarrow}_{\delta}(f)$ reveal the property δ. In order to guarantee that the transformer always characterizes the *minimal* revelation, $\mathcal{R}^{\uparrow}_{\delta}$ has to be monotone and extensive (approximating from above), and it has to be idempotent. Hence, $\mathcal{R}^{\uparrow}_{\delta}$ has to be an upper closure operator on the lattice $L \rightarrow L$, and dually $\mathcal{R}^{\downarrow}_{\delta}$ has to be a lower closure operator (*lco*). In the following, we consider the case when the revelation transformer do not trivially transform a function $f : L \rightarrow L$ with respect to a property $\delta \in uco(L)$. For this reason we find it convenient to introduce the notion of not-$\mathcal{R}$-trivial pair (f, δ).

Definition 3 (not-$\mathcal{R}$-trivial) *Given a function $f : L \rightarrow L$ on a complete lattice L and a property $\delta \in uco(L)$, we say that the pair (f, δ) is not-$\mathcal{R}$-trivial when: $\mathcal{R}^{\downarrow}_{\delta}(f) \neq \lambda x.x$ and $\mathcal{R}^{\uparrow}_{\delta}(f) \neq \lambda x.x$.*

The following result proves that $\mathcal{R}^{\uparrow}_{\delta}(f)$ and $\mathcal{R}^{\downarrow}_{\delta}(f)$ are precisely the minimal transformers inducing the revelation of the property δ. Namely, $\mathcal{R}^{\uparrow}_{\delta}(f)$ and $\mathcal{R}^{\downarrow}_{\delta}(f)$ are respectively an upper and lower closure operator and they satisfy (**).

Theorem 1 *Let L be a complete lattice, $f : L \rightarrow L$ and $\delta \in uco(L)$.*

1. *If δ is meet-uniform, then $\mathcal{R}^{\uparrow}_{\delta} \in uco(L \rightarrow L)$;*
2. *$\mathcal{R}^{\downarrow}_{\delta} \in lco(L \rightarrow L)$.*

In this theorem, we use the notion of uniformity, in particular of meet-uniformity which means that the greatest lower bound (glb) operation *preserves* the property δ, namely the glb of elements with same property δ has the same property δ. This precisely models the fact that we can find the best approximation of f from below sharing the same property δ of f. Thus, given a function $f : L \rightarrow L$, we have that $\mathcal{R}^{\uparrow}_{\delta}(f)$ returns the closest function that is greater than f and that reveals the property δ. For this reason, we refer to $\mathcal{R}^{\uparrow}_{\delta}(f)$ as the *minimal revelation (from above)* of f w.r.t. δ. Dually, we have that given a function $f : L \rightarrow L$, then $\mathcal{R}^{\downarrow}_{\delta}(f)$ returns the closest function that is smaller than f and that reveals the property δ. Analogously, we refer to $\mathcal{R}^{\downarrow}_{\delta}(f)$ as the *minimal revelation (from below)* of f w.r.t. δ.

4.1. Characterizing revelation transformers

The following result provides an explicit characterization of $\mathcal{R}^{\uparrow}_{\delta}(f)$, when the pair (f, δ) is not-$\mathcal{R}$-trivial. In the following, given a property $\delta \in uco(L)$ we define the *kernel* of δ with respect to an element $x \in L$ as $\mathtt{K}_{\delta}(x) \overset{\text{def}}{=} \{\, y \mid \delta(x) = \delta(y) \,\}$. Then we use the shorthand $\mathtt{K}^{\wedge}_{\delta}(x) \overset{\text{def}}{=} \wedge \{\, y \mid \delta(x) = \delta(y) \,\}$, and $\mathtt{K}^{\vee}_{\delta}(x) = \vee \{\, y \mid \delta(x) = \delta(y) \,\}$. Note that $\mathtt{K}^{\vee}_{\delta}(x) = \delta(x)$, since δ is an upper closure operator.

Theorem 2 *Let L be a complete lattice, $f : L \rightarrow L$ and $\delta \in uco(L)$ such that the pair (f, δ) is not-$\mathcal{R}$-trivial. We have that:*

1. *If δ is meet-uniform then $\mathcal{R}^{\uparrow}_{\delta}(f) = \lambda x.\, \mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)$;*
2. $\mathcal{R}^{\downarrow}_{\delta}(f) = \lambda x.\, \delta(x) \wedge f(x)$.

The above characterization says that the minimal revelation from above w.r.t. δ of f is the function that associates with each x the least upper bound between $f(x)$ and the smallest element that preserves δ on x. Indeed, this corresponds to add the minimal amount of information to $f(x)$ in order to make it preserve the property δ on x. An analogous reasoning holds for the minimal revelation from below.

Finally, we observe that by definition of $\mathcal{R}^{\uparrow}_{\delta}(f)$ and of $\mathcal{R}^{\downarrow}_{\delta}(f)$, when $\delta(x) = \delta(f(x))$ we have that $f(x) \in \{y \in C \mid f(x) \leq y, \delta(x) = \delta(y)\}$ and therefore $\mathcal{R}^{\uparrow}_{\delta}(f)(x) = f(x)$, and analogously $f(x) \in \{y \in C \mid y \leq f(x), \delta(x) = \delta(y)\}$ which means that $\mathcal{R}^{\downarrow}_{\delta}(f)(x) = f(x)$.

4.2. Examples of Revelations

In this section, we discuss two examples of existing transformations that are indeed revelations. In particular, a property is a revelation, namely it is the most abstract transformation preserving the property itself, and slicing is a revelation, namely it is the most abstract subprogram preserving the criterion. We can show that both are instances of the minimal revelation from below of the function $\lambda x.\top$.

Property as Revelation. First of all, let us observe that the minimal revelation from below of the function $\lambda x.\top$ w.r.t. δ, i.e., $\mathcal{R}^{\downarrow}_{\delta}(\lambda x.\top)$, is δ itself.

Proposition 1 $\mathcal{R}^{\downarrow}_{\delta}(\lambda x.\top) = \delta$.

Slicing as Revelation. In this section, we show that, by changing the interpretation of the underlying domain and by computing the minimal revelation from below of the function $\lambda x.\top$, we obtain *slicing*. Slicing is a program transformation technique that extracts from programs sets of commands that are relevant for a particular behavior, called the *criterion* [41]. A slicing criterion specifies the variables of interest and the program points where these variables have to be observed. Several different kinds of slicing have been collected in a unique framework [3] which has been recently enriched with the semantic (abstract) form of slicing [37, 38]. In this framework, slicing criteria are seen as projections on the trace semantics, and therefore are abstractions of the semantics. In this sense, we can observe that slicing produces the most abstract subprogram *preserving* the slicing criterion. Let $\mathcal{SC} \in uco(\wp(\Sigma^{\infty}))$ be the abstraction corresponding to the slicing criterion, for instance in standard backward static slicing, it is the abstraction observing only the output values of the criterion variables. By Proposition 1 we have that $\mathcal{R}^{\downarrow}_{\mathcal{SC}}(\lambda x.\top) = \mathcal{SC}$. This means that, the abstraction corresponding to the slicing criterion is the *semantics* of the desired slice. At this point, we can use the Cousot and Cousot program transformer framework for characterizing the corresponding program slicing (see Section 3 and [12]). In particular, the idea is to look for the simplest program, i.e., containing the smallest number of instructions, with the given semantics, in the set of all the *subprogram* of $\mathtt{P}$. Namely, we do not consider the whole set of all programs *Prog*, but only the subset of all the subprograms of $\mathtt{P}$, i.e., $\mathit{Prog}_{\mathtt{P}} \overset{\text{def}}{=} \left\{ \mathtt{Q} \in \mathit{Prog} \,\middle|\, \mathtt{Q} \text{ subprogram } \mathtt{P} \right\}$. Next example shows a simplification of the problem in order to provide an intuition of the relation between revelation and slicing. Namely, we show that the semantics of the slice of $\mathtt{P}$ for a given criterion is precisely the revelation preserving the criterion, computed on the lattice of subprograms of $\mathtt{P}$.

Example 3 *Consider the program* original *in Fig.* 1*. Suppose that the slicing criterion is the variable* nl *at the end of the program execution, which can be modeled by the abstraction of traces taking all the traces providing the same output value for* nl, $\mathcal{SC}([\![\text{original}]\!]) = \left\{ \tau \,\middle|\, \exists\tau' \in [\![\text{original}]\!].\, \tau_{\dashv}(nl) = \tau'_{\dashv}(nl) \right\}$*, where $\tau_{\dashv}(x)$ denotes the output value for variable x of a trace τ. Hence, the semantics corresponding to this criterion, which is the semantics of the slice, is precisely the set of all the traces computing the same final value for* nl*, namely $\mathcal{R}^{\downarrow}_{\mathcal{SC}}(\lambda x.\top) = \mathcal{SC}$. At this point, the simplest subprogram of* original *that has* *this as semantics is the computable slice w.r.t.* nl*, namely it is:*

```
or-slice-nl(){
      1·int c, nl = 0;
      3·while ((c = getchar()) != EOF){
            7·if(c == '\n')nl++;
            }
8·out(nl); }
```

4.3. Revelation vs Semantic program obfuscation

It is interesting to observe how the minimal revelation from below is related to the notion of semantic obfuscation in [14, 16]. The semantic notion of obfuscation of [14, 16] characterizes the obfuscating behavior of a semantic transformation $T : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ in terms of the most concrete property $\delta_T \in uco(\wp(\Sigma^\infty))$ that it preserves on program semantics. Thus, it is not surprising that this characterization relates to the revelation transformer that minimally modifies transformation T in order to preserve property δ_T. In particular, since by hypothesis δ_T is preserved by T, we have that $\mathcal{R}^{\downarrow}_{\delta_T}(T) = \mathcal{R}^{\uparrow}_{\delta_T}(T) = T$, which means that the minimal revelation of the code obfuscation T for property δ_T is exactly the obfuscation T itself.

Moreover, if we focus only on the property δ_T that an obfuscation T preserves we can maximize the obfuscating behavior by computing the revelation w.r.t. the property δ_T from the top semantic function $\lambda X.\Sigma^\infty$. Indeed, consider a semantic transformation $G : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ such that the most concrete property that it preserves is δ_T, namely such that $\delta_G = \delta_T$. Since G preserves δ_T and since $\mathcal{R}^{\downarrow}_{\delta_T}$ is monotone, we have that $G = \mathcal{R}^{\downarrow}_{\delta_T}(G) \sqsubseteq \mathcal{R}^{\downarrow}_{\delta_T}(\lambda X.\Sigma^\infty)$. Observe that all the transformations $G : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ such that $\delta_G = \delta_T$ and $\mathcal{R}^{\downarrow}_{\delta_T}(\lambda X.\Sigma^\infty)$ actually hide the same set of semantic program properties $\left\{ \varphi \in uco(\wp(\Sigma^\infty)) \,\middle|\, \exists X \in \wp(\Sigma^\infty),\ \delta_t(X) \not\subseteq \varphi(X) \right\}$. Hence, we can say that $\mathcal{R}^{\downarrow}_{\delta_T}(\lambda X.\Sigma^\infty) = \delta_T$ is the maximal transformation strategy, with respect to point-wise ordering of functions having the considered obfuscating behavior.

5. Modeling Concealment

In this section, we define and investigate the existence of the semantics transformers that *maximally* modify a program semantics (in the function point-wise order), in order to conceal a given property. It is maximal since it adds all the possible noise to the program semantics in order to confuse the property to protect, while keeping enough, of the original functionality, to recover the original preserved program semantics by revelation. This constraint is fundamental for modeling obfuscation when combined with revelation.

Formally, we first provide the definition of the concealment transformers on generic functions, and then we prove some of their interesting properties. Afterwards, we provide a characterization of the concealment transformers that shows how their construction is based on the property to be concealed. Finally, we discuss how the concealment transformers relate to the semantics-based notion of code obfuscation. The idea is to find the farthest functions from f, on the lattice $L \to L$, that conceals a certain property φ, and these are characterized as the farthest functions among all the functions having the same revelation transformer of f w.r.t. φ. These transformers are precisely the adjoints [30] of the revelation ones, which while transforming, in order not to reach the top, keep the strong bind with the original function consisting in having the same revelation transformer. The following definition formalizes these transformers maximally concealing a certain given property.

Definition 4 (Maximal concealment) *Let L be a complete lattice, $f : L \to L$ and $\varphi \in uco(L)$. We define $\mathcal{C}^{\downarrow}_{\varphi}, \mathcal{C}^{\uparrow}_{\varphi} : (L \to L) \to (L \to L)$ as:*

$$\mathcal{C}^{\downarrow}_{\varphi}(f) \overset{def}{=} \bigsqcap \left\{ g : L \to L \,\middle|\, \mathcal{R}^{\uparrow}_{\varphi}(f) = \mathcal{R}^{\uparrow}_{\varphi}(g) \right\}$$
$$\mathcal{C}^{\uparrow}_{\varphi}(f) \overset{def}{=} \bigsqcup \left\{ g : L \to L \,\middle|\, \mathcal{R}^{\downarrow}_{\varphi}(f) = \mathcal{R}^{\downarrow}_{\varphi}(g) \right\}$$

This means that $\mathcal{C}^{\downarrow}_{\varphi}(f)$ is the smallest among all the functions sharing the same revelation, from above w.r.t. φ of f, while $\mathcal{C}^{\uparrow}_{\varphi}(f)$ is the biggest among all the functions having the same revelation from below w.r.t. φ of f. These transformers are interesting if they provide, as result, a function that has the same revelation, and this clearly is not always true. In particular, the maximal concealments are defined as the adjoints of the revelation transformers and these adjoint transformers do not always exists. We know that an upper closure admits adjoint, which is a lower closure [30], if it is meet-uniform [23,36], while, dually, a lower closure admits adjoint, which is an upper closure, if it is join-uniform. It is worth noting that uniformity is a *local* property, namely a function g may be uniform on a particular input x, i.e., $g(x) = g(\wedge \{ y \,|\, g(x) = g(y) \})$, while failing uniformity on other inputs. In this case, we say that g is (meet)-uniform on x and w.r.t. x we can find the adjoint, i.e., $\bigwedge \{ y \,|\, g(x) = g(y) \}$. Hence, we have that if $\mathcal{R}^{\uparrow}_{\varphi}$, on a function f, is meet-uniform, then $\mathcal{C}^{\downarrow}_{\varphi}(f)$ is the minimum, meaning that it is an upper closure operator and behaves as adjoint of $\mathcal{R}^{\uparrow}_{\varphi}$, and dually if $\mathcal{R}^{\downarrow}_{\varphi}$ is join-uniform on f, then $\mathcal{C}^{\uparrow}_{\varphi}(f)$ is the maximum. For this reason, we have studied the properties of meet and join uniformity, respectively, of $\mathcal{R}^{\uparrow}_{\varphi}$ and $\mathcal{R}^{\downarrow}_{\varphi}$.

Theorem 3 *Let L be a complete Boolean algebra, $\varphi \in uco(L)$ and $f : L \to L$ such that the pair (f, φ) is not-$\mathcal{R}$-trivial. We have that:*

1. *If φ is meet-uniform then $\mathcal{R}^{\uparrow}_{\varphi} \in uco(L \to L)$ is meet-uniform on f;*
2. *$\mathcal{R}^{\downarrow}_{\varphi} \in lco(L \to L)$ is join-uniform on f.*

5.1. Characterizing concealment transformers

As done for the revelation transformers in Theorem 2, the following results provide a characterization of the maximal concealment transformers in terms of the kernel of the property that we want to conceal.

Theorem 4 *Let L be a complete Boolean algebra, $\varphi \in uco(L)$ and $f : L \to L$ such that the pair (f, φ) is not-$\mathcal{R}$-trivial. We have that:*

1. *If φ is meet-uniform then $\mathcal{C}^{\downarrow}_{\varphi}(f) = \lambda x.\ \bigvee \{ z \,|\, z \leq f(x),\ z \wedge \mathrm{K}^{\wedge}_{\varphi}(x) = \bot \}$;*
2. *$\mathcal{C}^{\uparrow}_{\varphi}(f) = \lambda x.\ \bigwedge \{ z \,|\, z \geq f(x), z \vee \varphi(x) = \top \}$;*

This means that, the maximal concealment from below w.r.t. property φ of f, is the function that associates, with each element x, the greatest element, smaller than $f(x)$, which provides no information about the property φ on x, namely the greatest element that is the complement, w.r.t. $\bot$, of the smallest element that preserves φ on x. Analogously, we have the dual characterization of the maximal concealment form above.

These characterizations turn out to be particularly meaningful when interpreted on a powerset domain ordered by set inclusion, as done in the following corollary.

Corollary 1 *Let D be a complete lattice, $L = \wp(D)$ and $\varphi \in uco(L)$. For each $f : L \leftarrow L$ such that the pair (f, φ) is not-$\mathcal{R}$-trivial. We have that:*

1. *If φ is meet-uniform then $\mathcal{C}^{\downarrow}_{\varphi}(f) = \lambda X.\, f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X)$;*
2. $\mathcal{C}^{\uparrow}_{\varphi}(f) = \lambda X.\, f(X) \cup (D \smallsetminus \varphi(X))$.

Indeed, on the powerset domain, we have that, for any given set $X \in \wp(D)$, the maximal concealment from below, namely $\mathcal{C}^{\downarrow}_{\varphi}(f)(X)$, can be obtained by erasing from $f(X)$ the smallest set in $\varphi(X)$. Hence, we erase the minimal information that leads to the property $\varphi(X)$. On the other hand, the maximal concealment from above, namely $\mathcal{C}^{\uparrow}_{\varphi}(f)(X)$, can be obtained by adding to $f(X)$ the biggest element in D that does not share the same property $\varphi(X)$. It is clear that in this way we add the maximal noise to $f(X)$ w.r.t. φ without any constraint on what we want to preserve/reveal.

5.2. *Concealment vs Code obfuscation.*

Let us understand the meaning of the concealment transformer, in the context of code obfuscation. To this end, we interpret the characterization given in Corollary 1 of the maximal concealment from above, in the code obfuscation scenario. Corollary 1 states that, given a semantic transformation $T : \wp(\Sigma^{\infty}) \rightarrow \wp(\Sigma^{\infty})$, its maximal concealment from above for property φ is given by $\mathcal{C}^{\uparrow}_{\varphi}(T) = \lambda X.T(X) \cup (\Sigma^{\infty} \smallsetminus \varphi(X))$. This means that given a program semantics $X \in \wp(\Sigma^{\infty})$ and a semantic program transformation T, if we want to characterize the maximal confusion added by T, with respect to property φ, we have to consider all the traces that do not belong to property $\varphi(X)$. Observe that, in this way, there is no guarantee of preserving any semantic property of the original semantics X. Thus, the maximal concealment from above corresponds to the maximal noise that can be added, w.r.t. a certain property φ, but it is not an obfuscation since this transformation does not consider the need to preserve the semantics of the original program to some extent.

Consider, for instance, CFGs whose definition is recalled in Section 2. In [21] the authors showed that CFGs can be constructed as abstractions of the program semantics, hence let $\mathcal{G} \in uco(\wp(\Sigma^{\infty}))$ be the abstraction such that $\mathcal{G}(\llbracket \mathtt{P} \rrbracket)$ is the CFG of the program $\mathtt{P}$. According to Corollary 1, the concealment transformation $\mathcal{C}^{\uparrow}_{\mathcal{G}}(id)$, starting from the identity, takes the semantics $\wp(\Sigma^{\infty}) \smallsetminus \mathcal{G}(\llbracket \mathtt{P} \rrbracket)$, namely the maximal semantics (containing as much noise as possible) whose corresponding CFG is different from P's CFG, and then adds the semantics of $\mathtt{P}$ for being extensive. It is clear that this semantics corresponds to a program that still may behave like $\mathtt{P}$, but that can also show many other different behaviors. This is because the concealment $\mathcal{C}^{\uparrow}_{\mathcal{G}}$ simply characterizes the action of adding confusion and not the *binding*, in terms of the observational property to preserve, that we aim to keep with the original semantics of $\mathtt{P}$.

The concealment provides an important understanding of what we have to add in order to *obfuscate* a given property: it characterizes the pool of all the possible computations that we may add in order to gain confusion on the observation of φ, hence concealing φ. At this point, it is worth noting that the revelation transformer for δ would allow us to refine this information by *avoiding* all the computations that do not preserve a property δ.

6. Characterizing property-driven obfuscations

In this section, we combine revelation and concealment transformers for modeling maximal obfuscation strategies in the semantics-based code obfuscation framework. In this way, we aim at improving the global understanding of code obfuscation. In our formalization, we clearly separate the property to conceal from the property to reveal, and this leads to a better comprehension of the interplay of these two properties on the semantics of the programs we want to obfuscate.

As discussed in Section 3, in the code obfuscation scenario, by combining revelation and concealment we can characterize an obfuscation starting from the specification of what we want to reveal, i.e., $\delta \in uco(\wp(\Sigma^\infty))$, and of what we want to conceal, i.e., $\varphi \in uco(\wp(\Sigma^\infty))$. The idea is to start from the identity function and maximally transform it in order to conceal the property that we want to protect and then to minimally transform the result in order to preserve the property that we want to reveal.

Definition 5 (Maximal property-driven obfuscation strategy) *Given $\delta, \varphi \in uco(\wp(\Sigma^\infty))$. The maximal property-driven obfuscation strategy for revealing δ and concealing φ is defined as follows:*

$$\mathfrak{O}^\delta_\varphi \overset{def}{=} \mathcal{R}^\downarrow_\delta(\mathcal{C}^\uparrow_\varphi(id))$$

where $id : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ is the identity function over the semantics of programs.

The following result provides a characterization of the maximal obfuscation strategy where we make more explicit its dependency on the properties, to conceal and to reveal. This is a direct consequence of the characterizations of concealment and revelation presented in the previous sections.

Proposition 2 $\mathfrak{O}^\delta_\varphi = \lambda X.\, X \cup (\delta(X) \cap (\Sigma^\infty \smallsetminus \varphi(X)))$.

The above result shows that, in order to conceal φ and to reveal δ on X, we have to start from the identity, and first add noise to the information concerning $\varphi(X)$ by adding those traces with a different φ property w.r.t. X, and then we guarantee the preservation of $\delta(X)$ by selecting only those traces having the same δ property of X. Finally, we add all the original traces in order to guarantee that the original semantics is preserved.

Coherently with what is proved in [14, 16], we can have a property-driven obfuscator for a given semantics X that reveals δ and conceals φ, only if $\delta(X)$ does not imply (namely reveal) $\varphi(X)$. Observe that when $\mathfrak{O}^\delta_\varphi(X) = X$ the obfuscating transformation is acting as the identity function and therefore it is not protecting property φ of the semantics X. By definition we have that $\mathfrak{O}^\delta_\varphi(X) \neq X$ if and only if $\delta(X) \cap (\Sigma^\infty \smallsetminus \varphi(X)) \neq \emptyset$, namely if $\delta(X) \not\subseteq \varphi(X)$. This means that we cannot hide, and therefore protect, anything that is implied by what we reveal, as formally stated in the following corollary.

Corollary 2 $\mathfrak{O}^\delta_\varphi(X) \neq X$ *if and only if* $\delta(X) \not\subseteq \varphi(X)$.

We can observe that the characterization of $\mathfrak{O}^\delta_\varphi$ provided in Proposition 2 is quite strong in the context of code obfuscation because it adds the whole semantics $[\![\mathtt{P}]\!]$ to the semantics of the obfuscated program. This implies that the original semantics has to be *contained* in the obfuscated program, which is a strong requirement. Indeed, since we have to reveal only δ, it is sufficient that the obfuscated program contains, i.e., preserves, only the *abstract* semantics $\delta([\![\mathtt{P}]\!])$. This observation is important also to partially fill the gap between the proposed strategy and existing code obfuscations which *transform* also traces of $\mathtt{P}$, namely obfuscations where the semantics of the original programs are not contained in the semantics of the obfuscated programs, i.e., such that $[\![\mathtt{P}]\!] \not\subseteq [\![\mathbb{T}(\mathtt{P})]\!]$. We observe that, the obfuscating component of $\mathfrak{O}^\delta_\varphi$ is the set we add to X, namely $\delta(X) \cap (\Sigma^\infty \smallsetminus \varphi(X))$, while the preservation condition forces any obfuscated version of X to stay inside $\mathtt{K}_\delta(X)$. Hence, to obfuscate $X \in \wp(\Sigma^\infty)$ for revealing δ and concealing φ, means to transform X into a set of traces that φ sees different from X, namely that does not belong to $\mathtt{K}_\varphi(X)$, while δ sees equivalent to X, namely that belongs to $\mathtt{K}_\delta(X)$. This observation leads to the following weakened characterization of property-driven code obfuscation strategies.

Definition 6 (Weakened property-driven obfuscation strategy) *Let $\varphi, \delta \in uco(\wp(\Sigma^\infty))$. $\hat{\mathfrak{O}}^{\delta}_{\varphi}(X)$ is a weakened property-driven obfuscation strategy of X that conceals φ and reveals δ if and only if*

$$\hat{\mathfrak{O}}^{\delta}_{\varphi}(X) \in \mathtt{K}_{\delta}(X) \cap (\wp(\Sigma^\infty) \smallsetminus \mathtt{K}_{\varphi}(X))$$

Hence, in this case the *maximal* property-driven obfuscation strategy is the maximal element of $\mathtt{K}_{\delta}(X)$ contained in $\wp(\Sigma)^* \smallsetminus \mathtt{K}_{\varphi}(X)$. Finally, the next result shows the relation between the existence condition of also this weakened version of property-driven obfuscation strategies and the semantic code obfuscation characterization provided in [14, 16] (see Section 3).

Corollary 3 *A weakened property-driven obfuscation strategy $\hat{\mathfrak{O}}^{\delta}_{\varphi}(X)$ exists if and only if*

$$\exists Y \in \wp(\Sigma^\infty).\ \delta(Y) \not\subseteq \varphi(Y).$$

The next example shows a simple program where we want to conceal the property of signs and we want to preserve the I/O semantics. Moreover, in this example, we show how we can model a real code transformation by means of the weakened property-driven obfuscation strategy.

Example 4 *Let $Sign^{+}(\wp(\mathbb{Z})) = \{\top, 0{+}, 0, 0{-}, +, -, \varnothing\}^2$ be the property of signs that can be lifted on $\wp(\Sigma^\infty)$ as follows: given $X \in \wp(\Sigma^\infty)$ we define $Sign^{+}(X) \overset{def}{=} \cup \{\tau \in \Sigma^\infty \mid \forall i.\ \tau_i \in Sign^{+}(\{\ \tau'_i \,|\, \tau' \in X\ \})\}$, where we recall that τ_i is the i-th state of the trace τ. Let $\mathfrak{I}$ be the I/O property. Let us consider, $\varphi = Sign^{+}$, $\delta = \mathfrak{I}$ and $X = [\![\mathtt{P}]\!] \in \wp(\Sigma^\infty)$, where $\mathtt{P}$ is given in Fig. 3. For simplicity, suppose $x \geq 0$. Then we have*

```
P :
  1. input $x$;
  2. $y := 2$;
  3. while $x > 0$ do
       4. $y := y + 2$;
       5. $x := x - 1$
     endw
  6. output $y$;
```

```
T(P) :
  1. input $x$;
  2. $y := -2$;
  3. while $x > 0$ do
       4. $y := y + 2$;
       5. $x := x - 1$
     endw
  6'. if $x = 0$ then $y := y + 4$; else skip fi
  6. output $y$;
```

Fig. 3. Original and obfuscated program of Example 4

that

$$\begin{aligned} Sign^{+}([\![\mathtt{P}]\!]) &= \{\ \tau \,\big|\, \exists n \geq 0.\ \tau \in \langle 0{+}, \bot\rangle \to \langle 0{+}, +\rangle \to \langle +, +\rangle^n \to \langle 0, +\rangle\ \} \\ \mathfrak{I}([\![\mathtt{P}]\!]) &= \{\ \tau \,\big|\, \exists \tau' \in [\![P]\!].\ \tau_{\dashv} = \tau'_{\dashv},\ \tau_{\vdash} = \tau'_{\vdash}\ \} \\ \mathtt{K}_{\mathfrak{I}}([\![\mathtt{P}]\!]) &= \{\ Y \,\big|\, \mathfrak{I}(Y) = \mathfrak{I}([\![P]\!])\ \} \end{aligned}$$

Then, an obfuscation $\mathbb{T}(\mathtt{P})$, following the weakened property-driven strategy $\hat{\mathfrak{O}}^{\mathfrak{I}}_{Sign^{+}}([\![\mathtt{P}]\!])$, has to be such that $\mathbb{T}(\mathtt{P}) \in \mathtt{K}_{\mathfrak{I}}([\![\mathtt{P}]\!])$ and $\mathbb{T}(\mathtt{P}) \in \wp(\Sigma^\infty) \smallsetminus \mathtt{K}_{Sign^{+}}([\![\mathtt{P}]\!])$. Hence, the semantics of the obfuscation must be a set of traces sharing the same I/O behavior with $\mathtt{P}$, but whose sign property changes. At this point

[2] where the abstract element $+$ represents the set of positive numbers without 0, and $-$ the set of negative numbers without 0

any program whose semantics satisfies these conditions may be considered as an obfuscation of $\mathtt{P}$. *For instance, consider* $\mathbb{T}(\mathtt{P})$ *on the right in Fig.* 3, *we have* $\mathfrak{I}([\![\mathtt{P}]\!]) = \mathfrak{I}([\![\mathbb{T}(\mathtt{P})]\!])$ *and*

$$\mathit{Sign}^{+}([\![\mathbb{T}(\mathtt{P})]\!]) = \left\{\, \tau \,\middle|\, \exists n.\, \tau \in \langle 0+, \bot\rangle \to \langle 0+, -\rangle \to \langle +, 0+\rangle^{n} \to \langle 0, +\rangle \,\right\} \neq \mathit{Sign}^{+}([\![\mathtt{P}]\!]).$$

7. Applicability

In this section, we provide an example of how our approach can be instantiated. It is worth noting that what we provide is not a particular obfuscation technique, but a framework for understanding existing obfuscations and for designing new ones depending on two properties: what we want to conceal and what we want to leave unchanged, namely to reveal of the original program semantics. In this way, as we will also explain later in Section 7.2, several existing techniques may be characterized precisely in terms of what they preserve and what they protect, but moreover we can even understand better how they work. For instance, the use of opaque predicates, which are used for adding fake branching in the CFG, confuses (and therefore protects) the control structure of the CFG but it releases (namely preserves) more than the I/O semantics, since the whole concrete executions are preserved.

As far as the design of new techniques is concerned, our framework can be used to understand, once we fix the two properties (φ to protect and δ to preserve), what kind of behaviors we have to add to the semantics of the program, and therefore how we have to transform the program, in order to guarantee that we change φ and we leave unchanged δ.

In the next section, we present recent work on obfuscation based on language interpreters. This is a particular technique that can be used whenever we aim at revealing the program semantics, in the sense that by construction the interpreter-based obfuscation always preserves the I/O program behavior. At the same time, this technique transforms the program code by including some syntactic fragments that are known for making imprecise the analysis we aim at concealing. The reason why this approach easily fits in our framework is that it is based on the same model of attacker. It is clear that, as we underlined more than once, this is an obfuscation technique working precisely for the intended model of attacker, namely parametric on the property we aim at concealing. In particular, this means that if we face an attacker able to perform less precise analyses, then surely our program is protected, while if we face an attacker performing more precise analyses, then it may be able to disclose (partial) information about the property we aim at concealing.

7.1. Property-driven obfuscation and distorted interpreters

Standard obfuscation aims at revealing precisely the observable program behavior while concealing other properties, such as the data dependencies or the control flow structure. In these cases, namely when we do not want to reveal anything different, more precise or more approximated than the observable semantics, we may use the obfuscation technique based on the design of distorted interpreters [21]. The aim, of this approach, is to obfuscate $\mathtt{P}$ either by making its observation hard and imprecise, or by making protected information imperceptible.

In this section, we propose a systematic technique for building an obfuscator protecting/concealing a given property, when the property to reveal is simply the I/O behaviour. The idea, is that of transforming the original program $\mathtt{P}$ into an obfuscated program $\mathtt{P}'$ by *specializing a distorted (but still correct) interpreter* as in [31, 32]. Let us explain the idea on an example [21, 25]. Let $\mathtt{P}$ be the simple command, $\mathtt{C} : \mathtt{x} = \mathtt{a} * \mathtt{b}$, multiplying $\mathtt{a}$ and $\mathtt{b}$, and storing the result in $\mathtt{x}$, as considered in [20]. An automated program sign analysis approximating values in the simple domain of signs is clearly able to catch, with no loss of precision, the intended behavior of C with respect to sign, as the sign abstraction $\mathit{Sign} = \{\bot, +, 0, -, \top\}$, is complete for integer multiplication. However, if we obtain the obfuscated program $\mathtt{P}' = \mathbb{T}(\mathtt{P})$ by replacing C with:

```
P' : ⎧ ^{1.} input $a, b$;
     ⎪ ^{2.} $x := 1$;
     ⎪ ^{3.} if $b \le 0$
     ⎪       then ^{4.} $a := -a$; ^{5.} $b := -b$ else skip fi
     ⎨ ^{6.} while $b \neq 0$ do
     ⎪         ^{7.} $x := a + x$; ^{8.} $b := b - 1$
     ⎪       endw
     ⎩ ^{9.} output $x$;
```

The sign analysis is now unable to extract any information concerning the computed sign, because the rule of signs is incomplete for integer addition. In particular, we observe that when $a = -$ and $b = +$, then $x = +$ (being $\mathit{Sign}(1) = +$) and $a + x$ is the sum of a negative number with a positive one, and therefore we lose the sign information. $\mathbb{T}(\mathtt{P})$ is therefore an obfuscation of P for the attack performed by sign analysis.

The main idea of this work is to obfuscate by *specializing* an *obfuscating self-interpreter*. An interpreter executes programs written in (potentially) another programming language. The interpreter is *self* if it is written precisely in the interpreted language. This choice is connected with the process of specialization. Indeed, specialization (better known as partial evaluation) means to partially evaluate a program on some known inputs, namely the specialized program [32] is precisely the same but with some instantiated computations. Hence, an interpreter specialized on a program is precisely the interpreter where all the computations concerning the input program are instantiated. Now, suppose that the program to obfuscate is written in the language $\mathfrak{L}$, then if we want the obfuscated program (i.e., the specialized interpreter) to be written in the same language $\mathfrak{L}$, then the interpreter has to be written precisely in $\mathfrak{L}$, namely it has to be a self-interpreter[3]. Hence, $\mathbb{T}(\mathtt{P})$ can be obtained by modifying a self-interpreter `interp` in order to force abstract interpretation to deal with operations that induce incompleteness in the attacker. Let us explain the technique on the sign example.

Suppose the distorted interpreter $\mathtt{interp}^{+}$ recursively implements multiplication by a sequence of additions in the obvious way (as above). This yields, by specialization, a modified program $\mathtt{P}' := [\![\mathtt{spec}]\!](\mathtt{interp}^{+}, \mathtt{P})$ which is precisely $\mathbb{T}(\mathtt{P})$. Moreover, in this work it has been easily observed that, in general, if $\mathtt{P}' := [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})$ is the result of specializing a self-interpreter to program P, then it is immediate that $[\![\mathtt{P}]\!] = [\![\mathtt{P}']\!]$, by simple equational reasoning [21]. Therefore function $\lambda\mathtt{P}.[\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})$ is a semantics-preserving program transformer [31]. However, even though P and $\mathtt{P}'$ are *semantically* equivalent, they may be *syntactically* quite different (far more different than just by renaming). In particular [31]:

1. program $\mathtt{P}'$ inherits the *programming style* of `interp`; but
2. program $\mathtt{P}'$ inherits the *algorithm* of program P;.

The reason for Point 1 is that program $\mathtt{P}'$ is specialized code taken from the interpreter `interp`: $\mathtt{P}'$ consists of the operations of `interp` that depend on its dynamic input d (and not only on P, else

[3] It is worth noting, anyway, that this is not mandatory in general, if we admit as part of the obfuscation process the possibility of changing also the programming language.

```
input $\texttt{P}, d$;                         Program to be interpreted, and its data
$pc := 2;\ store := [in \mapsto d, out \mapsto 0, x_1 \mapsto 0, \ldots]$;
 while $pc < length(\texttt{P})$ do
     $instruction := lookup(\texttt{P}, pc)$;       Find the $pc$-th instruction
     case $instruction$ of                           Dispatch on syntax
     skip    : $pc := pc + 1$;
     $x := e$ : $store := store[x \mapsto eval(e, store)];\ pc := pc + 1$;
     ... endw ;
output $store[out]$;
$eval(e, store)$ = case $e$ of          Function to evaluate expressions
     $constant$ : $e$
     $variable$ : $store(e)$
     $e1 + e2$  : $eval(e1, store) + eval(e2, store)$
     $e1 - e2$  : $eval(e1, store) - eval(e2, store)$
     $e1 * e2$  : $eval(e1, store) * eval(e2, store)$
     ...
end
```

Fig. 4. Simple self-interpreter

they would have been "specialized away"). A *general-purpose program transformer* can thus be built by programming self-interpreter `interp` in a style appropriate to the desired transformation. In this style we can embed the property we aim to conceal. In the example above, the specialized interpreter can transform multiplication to sequences of additions because we aim to conceal the sign property for which the addition become imprecise. The design of the distorted interpreter starting from the property to conceal is still made ad hoc, but it is an ongoing work based on the analysis of the strong relation between obfuscation and incompleteness [24].

The reason for Point 2 is that, even though program $\texttt{P}'$ may be a disguised form of $\texttt{P}$, a correct interpreter `interp` must faithfully execute the operations that $\texttt{P}$ specifies. In usual practice, the transformed (by the specialized interpreter) program $\texttt{P}'$ will perform *the same computations in the same order* as those performed by $\texttt{P}$, guaranteeing the sound revelation of the program semantics.

The overall structure of the interpreter is traditional, with a "dispatch on syntax" loop: find the form of the current $\texttt{P}$ instruction at *pc* (program counter), and then execute it. The memory of program $\texttt{P}$ is held in `interp` variable *store* [21]. In Fig. 4 is provided the algorithm for of a simple self-interpreter (implemented in SCHEME for experiments [21]). Assume input program $\texttt{P}$ has variables $in, out, x_1, \ldots, x_n$. An example of specialization of this self interpreter is provided in Fig. 5 explained in Section 7.2.

7.2. Validating Examples

In the following, we model three well-known examples of code obfuscation in our framework of revelation and concealment. In particular, we show how the framework proposed characterizes a pool of possible transformations that satisfy the fixed constraints of revelation and concealment. The following examples show that, whenever we want to preserve the behavior of programs and we are able to associate to an obfuscation strategy a family of syntactic elements that implement the obfuscation, then we can use the distorted interpreters introduced in the previous section [21], for choosing one precise element from the characterized pool.

```
input $\mathtt{P}, d$;                    Program to be interpreted, and *its* data
$pc := 2;\ store := [in \mapsto obf(d), out \mapsto obf(0), x_1 \mapsto obf(0), \ldots]$;
 while $pc < length(\mathtt{P})$ do
      $instruction := lookup(\mathtt{P}, pc)$;
      case $instruction$ of                            Dispatch on syntax
      skip      : $pc := pc + 1$;
      $x := e$ : $store := store[x \mapsto eval(e, store)];\ pc := pc + 1$;
      ... endw ;
output $dob(store[out])$;
$obf(x) = x * 2;\ dob(x) = x/2$          Obfuscation/de-obfuscation
$eval(e, store)$ = case $e$ of
      $constant : obf(e)$                          Obfuscate constants
      $variable : store(e)$
      $e1 + e2 : obf(dob(eval(e1, store)) + dob(eval(e2, store)))$
      $e1 - e2 : obf(dob(eval(e1, store)) - dob(eval(e2, store)))$
      ...
end
```

Fig. 5. Self-interpreter for data-type obfuscation

Data-type Obfuscation. Let us consider, in a more detailed way, data-type obfuscation cited in the introduction, namely the obfuscation techniques based on the encoding of data [18]. In this case, obfuscation is achieved by data refinement, namely by exploiting the complexity of more complex data structures or values in such a way that actual computations can be viewed as abstractions of the refined (obfuscated) ones. The idea is to choose a pair of commands c^α and c^γ such that $c^\gamma; c^\alpha$ is equivalent to **skip**. This means that both c^α and c^γ are commands of the form: $c^\alpha \overset{\text{def}}{=} x := G(x)$ and $c^\gamma \overset{\text{def}}{=} x := F(x)$, for some function F and G. A program transformation $\mathbb{T}(\mathtt{P}) \overset{\text{def}}{=} c^\gamma; \xi_x(\mathtt{P}); c^\alpha$ is a data-type obfuscation for data-type x if $\mathbb{T}(\mathtt{P})$ and $\mathtt{P}$ have the same denotational behavior on x, where ξ_x adjusts the data-type computation for x on the refined type (see [18]). It is known that data-type obfuscations can be modeled as adjoint functions (Galois connections), where c^γ represents the program concretizing , viz. refining, the datum x and c^α represents the program abstracting the refined datum x back to the original data-type. As proved in [20], this is precisely modeled as a pair of adjoint functions: $\alpha : \mathbb{V} \longrightarrow \mathbb{V}^{\Re}$ and $\gamma : \mathbb{V}^{\Re} \longrightarrow \mathbb{V}$ relating the standard data-type $\mathbb{V}$ for x with its refined version $\mathbb{V}^{\Re}$. Consider the following program [21]: the small input program $\mathtt{P}$ to the left is obfuscated into the equivalent program $\mathbb{T}(\mathtt{P})$ on the right (where constant expressions have been evaluated):

```
1. input $x$;
2. $y := 2$;
3. while $x > 0$ do
      4. $y := y + 2$; 5. $x := x - 1$
   endw
6. output $y$;
```

```
1. input $x$;
2. $x := 2 * x$; 3. $y := 2 * 2$;            Obfuscate input $x$ and $y := 2$
4. while $x/2 > 0$ do           De-obfuscate $x$
      5. $y := 2 * (y/2 + 2)$; 6. $x := 2 * (x/2 - 1)$
   endw
7. output $y/2$;               De-obfuscate output
```

In this case, c^γ corresponds to the commands at the new lines 2 and 3 that multiply by 2 the values of x and y, then ξ_x and ξ_y are responsible for the adjustment of the other computations inside the program (for example in the guard of the while, the check is done on the value of x divided by 2), and c^{α} corresponds to the command at line 7 where the value of y is divided by 2 before giving it in output. Let $\mathit{Par}(\wp(\mathbb{Z})) = \{\top, \mathtt{ev}, \mathtt{od}, \varnothing\}$ be the property of parity that can be lifted on $\wp(\Sigma^{\infty})$ as previously seen in Example 4. Let $\mathcal{J}$ be the I/O property seen in Sect. 3.1. Let us consider, $\varphi = \mathit{Par}$, $\delta = \mathcal{J}$ and $X = [\![\mathtt{P}]\!] \in \wp(\Sigma^{\infty})$. On input $\langle 3, \bot\rangle$, where the first value is the value of x and the second value the value of y, we have that (in sake of simplicity we ignore repetition of states, i.e., $\langle a, b\rangle \to \langle a, b\rangle = \langle a, b\rangle$):

$$\begin{aligned}
\mathit{Par}([\![\mathtt{P}]\!]) &= \left\{\, \tau \,\middle|\, \tau \in \langle\mathtt{od}, \bot\rangle \to \langle\mathtt{od}, \mathtt{ev}\rangle \to \langle\mathtt{ev}, \mathtt{ev}\rangle \to \langle\mathtt{od}, \mathtt{ev}\rangle \to \langle\mathtt{ev}, \mathtt{ev}\rangle \,\right\}\\
\mathcal{J}([\![\mathtt{P}]\!]) &= \left\{\, \tau \,\middle|\, \exists \tau' \in [\![P]\!].\, \tau_{\dashv} = \tau'_{\dashv},\ \tau_{\vdash} = \tau'_{\vdash} \,\right\}\\
\mathtt{K}_{\mathcal{J}}([\![\mathtt{P}]\!]) &= \left\{\, Y \,\middle|\, \mathcal{J}(Y) = \mathcal{J}([\![P]\!]) \,\right\}
\end{aligned}$$

Then a property-driven obfuscation $\hat{\mathfrak{O}}^{\mathcal{J}}_{\mathit{Par}}([\![\mathtt{P}]\!])$ has to be such that $\hat{\mathfrak{O}}^{\mathcal{J}}_{\mathit{Par}}([\![\mathtt{P}]\!]) \in \mathtt{K}_{\mathcal{J}}([\![\mathtt{P}]\!])$ and that $\hat{\mathfrak{O}}^{\mathcal{J}}_{\mathit{Par}}([\![\mathtt{P}]\!]) \in \wp(\Sigma^{\infty}) \smallsetminus \mathtt{K}_{\mathit{Par}}([\![\mathtt{P}]\!])$. Hence, the semantics of the obfuscation must be a set of traces such that the I/O behavior is precisely that of $\mathtt{P}$, but whose *Par* property changes. At this point, any program whose semantics satisfies these conditions may be considered as an obfuscation of $\mathtt{P}$. For instance, consider $\mathbb{T}(\mathtt{P})$ above on the right, we have that the I/O property is preserved, since $\mathcal{J}([\![\mathtt{P}]\!]) = \mathcal{J}([\![\mathbb{T}(\mathtt{P})]\!])$, while the *Par* property is not:

$$\mathit{Par}([\![\mathbb{T}(\mathtt{P})]\!]) = \left\{\, \tau \,\middle|\, \exists n.\, \tau \in \langle\mathtt{od}, \bot\rangle \to \langle\mathtt{ev}, \mathtt{ev}\rangle^{n} \,\right\} \neq \mathit{Par}([\![\mathtt{P}]\!]).$$

Hence data-type obfuscation is a possible way to formalize a systematic technique for generating transformed code satisfying our obfuscation framework. Data-type obfuscation can be easily implemented by specializing an interpreter. In particular, for instance the interpreter automatically generating the obfuscation above [21] is provided in Fig. 5.

Opaque Predicates. Let us consider the opaque predicates obfuscation. Opacity is an obfuscation technique based on the idea of confusing the control structure of a program by inserting predicates that are always true (false) independently of the memory, or (unknown) which may lead only to identical paths [7]. It is clear that our approach allows us to characterize which property each code obfuscation aims at preserving.

In the case, namely by using opaque predicates, we observe that the transformation preserves something more than the simple I/O behavior. Let us consider the domain of control flow graphs (CFG) $\mathcal{G}$ (see Section 5.2), for each $\mathtt{G} \in \mathcal{G}$ let us denote by $\mathtt{Nodes}(\mathtt{G})$ and $\mathtt{Edges}(\mathtt{G})$ respectively its nodes and its edges. Let us define the following relation: Let $\mathtt{G}, \mathtt{G}' \in \mathcal{G}$, then we say that $\mathtt{G}$ is sub-graph of $\mathtt{G}'$, written $\mathtt{G} \prec \mathtt{G}'$, iff $\mathtt{Nodes}(\mathtt{G}) \subseteq \mathtt{Nodes}(\mathtt{G}')$ and $\mathtt{Edges}(\mathtt{G}) \subseteq \mathtt{Edges}(\mathtt{G}')$. At this point it is clear that the obfuscation based on the insertion of opaque predicates consists in adding nodes and edges without changing in other ways the original CFG. Hence, let us define the following properties:

$$\begin{array}{ll}
\mathcal{S}_u \in \mathit{uco}(\wp(\mathcal{G})) & \mathcal{S}_u \overset{\text{def}}{=} \lambda\mathbb{G} \in \wp(\mathcal{G}).\ \left\{\, \mathtt{G}' \in \mathcal{G} \,\middle|\, \exists \mathtt{G} \in \mathbb{G}.\, \mathtt{G} \prec \mathtt{G}' \,\right\}\\
\mathcal{J}_g \in \mathit{uco}(\wp(\mathcal{G})) & \mathcal{J}_g \overset{\text{def}}{=} \lambda\mathbb{G} \in \wp(\mathcal{G}).\ \left\{\, \mathtt{G}' \,\middle|\, \mathtt{Exec}_{\mathtt{I}}(\mathtt{G}) = \mathtt{Exec}_{\mathtt{I}}(\mathtt{G}') \,\right\}\\
 & \text{where } \mathtt{Exec}_{\mathtt{I}}(\mathtt{G}) \overset{\text{def}}{=} \left\{\, \langle \tau_{\vdash}, \tau_{\dashv}\rangle \,\middle|\, \tau \text{ is an execution of a path in } \mathtt{G} \,\right\}
\end{array}$$

In particular, we have that $\mathcal{S}_u(\mathtt{G})$ is the set of all graphs of which $\mathtt{G}$ is a sub-graph, and $\mathcal{J}_g(\mathtt{G})$ are the set of graphs that have the same set of executions of $\mathtt{G}$. Hence, we can say that the property that we want to preserve is $\delta_{op} = \mathcal{S}_u \sqcap \mathcal{J}_g$, which is additive by construction. Indeed, $\delta_{op}(\mathtt{G})$ precisely collects all the

graphs containing $\mathtt{G}$ and having the same set of executions of $\mathtt{G}$.
At this point, we aim at confusing the control flow graph, which is the domain we are starting from, hence the property φ we want to conceal is simply the identity. The following result is a slight rewriting of a theorem in [21].

Proposition 3 *Given* $\mathtt{G} \in \mathcal{G}$*, then we have that* $\mathtt{G}' \in \delta_{op}(\mathtt{G})$ *iff* $\mathtt{G}'$ *differs from* $\mathtt{G}$ *only for the insertion of opaque predicates (true, false or unknown).*

At this point note that, in order to obtain a weakened property-driven obfuscation of a control flow graph $\mathtt{G}$ we observe that

$$\begin{array}{l} \mathtt{K}_{id}(\mathtt{G}) = \ \{\mathtt{G}\} \\ \mathtt{K}_{\delta_{op}}(\mathtt{G}) = \left\{ \, \mathtt{G}' \,\middle|\, \delta_{op}(\mathtt{G}) = \delta_{op}(\mathtt{G}') \, \right\} = \delta_{op}(\mathtt{G}) \text{ (By additivity)} \end{array}$$

Hence, following Definition 6, a weakened property-driven obfuscation of a control flow graph G based on the insertion of opaque predicates is any obfuscating transformation in

$$\mathtt{K}_{\delta_{op}}(\mathtt{G}) \cap (\wp(\mathcal{G} \smallsetminus \mathtt{K}_{id}(\mathtt{G}))) = \mathtt{K}_{\delta_{op}}(\mathtt{G}) \smallsetminus \{\mathtt{G}\}$$

Namely, an obfuscation is precisely any CFG preserving the structure and the program semantics as determined by δ_{op}, but which is simply different from $\mathtt{G}$. This means that any strategy building such a graph starting from $\mathtt{G}$ is an obfuscation of $\mathtt{G}$ and it must contain opaque predicates by Proposition 3. For instance, the obfuscation technique proposed in [21] could be considered as a way for systematically building such an obfuscation. Namely, also in this case we could implement a syntactic transformation which can be implemented by means of an obfuscating interpreter that adds opaque predicates in specific program points[4].

Slicing Obfuscation. Consider the slicing obfuscation example with programs in Fig. 1. We already observed that $\mathcal{D}(\mathit{original}) \neq \mathcal{D}(\mathit{obfuscated})$ while $\mathcal{I}(\mathit{original}) = \mathcal{I}(\mathit{obfuscated})$. This means that there exists $\hat{\mathfrak{O}}^{\mathcal{I}}_{\mathcal{D}} \in \mathtt{K}_{\mathcal{I}}$ and $\hat{\mathfrak{O}}^{\mathcal{I}}_{\mathcal{D}} \in \wp(\Sigma^{\infty}) \smallsetminus \mathtt{K}_{\mathcal{D}}$ such that $[\![\mathit{obfuscated}]\!] = \hat{\mathfrak{O}}^{\mathcal{I}}_{\mathcal{D}}([\![\mathit{original}]\!])$.
As explained before, a technique has been proposed [21] which is based on the specialization of self-interpreters in order to obtain an obfuscator preserving $\mathcal{I}$, while protecting the dependency abstraction. This is obtained by transforming programs by adding fake dependencies. The $\mathcal{I}$ semantics is instead preserved by construction, since interpretation and specialization cannot change the program semantics.

8. Discussion and Future Work

In this paper, we provide a formal framework where functions can be maximally transformed in order to change a given property φ (concealment transformer), or can be minimally transformed preserving a given property δ (revelation transformer). We motivate this formal framework in the context of semantic code obfuscation, where indeed we have to make impractical the observation of information to protect while preserving the I/O functionality of a program. In particular, we combine the revelation and concealment transformers into a characterization of an obfuscating transformations based, precisely, on the properties φ and δ. We obtain, so far, a general characterization of a semantic code obfuscations which

[4] See [21] for more details.

allows us to further reason about the existing relations between the different actors occurring in the obfuscation scenario, and therefore to deeply understand their interplay. In particular, it may seem that, since we treat concealment and revelation properties separately, we are supposing that they are independent one from the other. Indeed, this is not the case, in fact, it is precisely the possible existing interaction between the property φ to protect and other program properties that induces us to combine a concealment transformation, transforming a program for protecting φ, with a revelation transformation recovering the functionality that could have been transformed as side effect of the concealment. In other words, a concealment transformer is a transformer that conceals *at least* the given property φ but which, being maximal, surely transforms many other program properties. Then it is revelation that allows us to recover at least the property we want to preserve. In this way, the noise that remain in the maximal obfuscation strategy is the pool of all computations that, while changing φ, preserves the desired functionality.

In the following, we discuss some further research directions arising form this work and that we plan to address in the future. By instantiating this characterization to a specific program P, we obtain a specification of the obfuscated version of the considered program. Namely, $\mathcal{R}^{\downarrow}_{\delta}(\mathcal{C}^{\uparrow}_{\varphi}(id))(\llbracket \mathtt{P} \rrbracket)$ represents the semantics of the obfuscation of program P. As future work, we plan to investigate the existence of this obfuscated program in terms of the interplay between the two considered properties φ and δ in the semantics of P. Our intuition is that, given a program P, it is possible to obfuscate it in order to reveal φ while concealing δ only if these two properties are somehow independent in the semantics of P. More specifically, we aim at formalizing the notion of independence of those two properties in terms of abstract non-interference [17,22].

In Section 6, we have defined the maximal property-driven obfuscation strategy as $\mathcal{R}^{\downarrow}_{\delta}(\mathcal{C}^{\uparrow}_{\varphi}(id))$, namely starting from the identity function on program semantics. It would be interesting to study what happens when we apply revelation and concealment to existing obfuscations. Consider an obfuscation $\mathbb{T} : Prog \rightarrow Prog$ and its semantic counterpart $T : \wp(\Sigma^{\infty}) \rightarrow \wp(\Sigma^{\infty})$. Then, $\mathcal{R}^{\downarrow}_{\delta}(T)$ minimally transforms obfuscation T in order to make it reveal property δ. Whereas, $\mathcal{R}^{\downarrow}_{\delta}(\mathcal{C}^{\uparrow}_{\varphi}(T))$ somehow optimizes obfuscation T to make it reveal δ and conceal φ. Moreover, it may be worth studying how different concealments can be composed, namely understanding the composition of abstractions distributes on the concealments. We plan to validate these observations on existing obfuscation techniques.

Another future research direction regards the investigation of the potency and resilience of the obfuscation transformers based on the proposed strategy, with respect to an attacker. In this work, we have implicitly assumed that the attacker is modeled by the property φ that we want to protect. In this way the transformation corresponding to $\mathfrak{O}^{\delta}_{\varphi}$ provides a characterization of the obfuscation that is able to defeat such an attacker. In general, we can think of an attacker as modeled by a further abstraction $\beta \in uco(\wp(\Sigma^{\infty}))$ of program semantics and then study the relations between the attacker β, the protected property φ and the released property δ in order to understand the efficiency of the proposed obfuscation with respect to β.

In the literature, there exists another formalization of the notion of obfuscation where the potency of the transformation is characterized on the abstract program semantics, by means of completeness[5], instead of on the abstraction of the concrete semantics, as we have done in this work. In particular, [21, 24, 25] provide a description of obfuscations as transformers preserving and protecting fixpoint abstract semantics and the potency of a program transformation $\mathbb{T}$ as the properties for which the transformation is *incomplete*, namely imprecise. In this case, a property is revealed if and only if it is revealed by

[5] Completeness is a well-known notion in abstract interpretation that characterizes program analysis precision [10].

each step of the fixpoint semantic computation. Hence, the idea is to generalize revelation transformers, and therefore concealments, by considering this more general notion of revelation instead of the simple abstraction of the whole semantics.

From the theoretical point of view, this can be related with other property-driven function transformers, such as the complete shells and cores [27], the incomplete transformers [24], the transformers towards additivity [1], obtaining so far a framework for property-driven transformers parametric on the property to guarantee.

Finally, we are clearly interested in designing and developing obfuscation techniques based on the proposed strategy and therefore parametric on the properties to reveal and to conceal.

Acknowledgments

We would like to thank Roberto Giacobazzi for the inspiring discussions at the beginning of this work. We are also grateful to the anonymous reviewers for their useful comments and suggestions.

References

[1] H. Andéka, R. J. Greechie, and G. E. Strecker. On residuated approximations. In *Categorical Methods in Computer Science*, volume 393 of *Lecture Notes in Computer Science*, pages 333 – 339, 1989.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.

[3] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program*, 62(3):228–252, 2006.

[4] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *ISC '01: Proceedings of the 4th International Conference on Information Security*, pages 144–155. Springer-Verlag, 2001.

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

[6] C. Collberg and C. D. Thomborson. Software watermarking: models and dynamic embeddings. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 311–324. ACM, 1999.

[7] C. Collberg, C. D. Thomborson, and D. Low. Manufactoring cheap, resilient, and stealthy opaque constructs. In *Proc. of Conf. Record of the 25st ACM Symp. on Principles of Programming Languages* (*POPL '98*), pages 184–196. ACM Press, 1998.

[8] K. D. Cooper and L. Torczon. *Engineering a Compiler (II Edition)*. Elsevier, 2012.

[9] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.

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

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

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

[13] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 301–310. IEEE Computer Society, 2005.

[14] 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.

[15] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. In *Proc. of the 11th Internat. Conf. on Algebraic Methodology and Software Technology* (*AMAST '06*), volume 4019 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, 2006.

[16] Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[17] Mila Dalla Preda, Isabella Mastroeni, and Roberto Giacobazzi. Analyzing program dependencies for malware detection. In *3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop PPREW14*, page 6. ACM, 2014.

[18] S. Drape, C. Thomborson, and A. Majumdar. Specifying imperative data obfuscations. In J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, editors, *ISC - Information Security*, volume 4779 of *Lecture Notes in Computer Science*, pages 299 – 314. Springer Verlag, 2007.

[19] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[20] R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*, pages 7–20. IEEE Press., 2008.

[21] R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In O. Kiselyov and S. Thompson, editors, *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, pages 63 – 72. ACM Press, 2012.

[22] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM-Press, 2004.

[23] R. Giacobazzi and I. Mastroeni. Transforming abstract interpretations by abstract interpretation. In M. Alpuente, editor, *Proc. of The 15th International Static Analysis Symposium, SAS'08*, volume 5079 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2008.

[24] R. Giacobazzi and I. Mastroeni. Making abstract interpretation incomplete - modeling the potency of obfuscation. In A. Miné and D. Schmidt, editors, *19th International Static Analysis Symp. (SAS '12)*, volume 7460 of *Lecture Notes in Computer Science*, pages 129 – 145, 2012.

[25] R. Giacobazzi, I. Mastroeni, and M. Dalla Preda. Maximal incompleteness as obfuscation potency. *Formal Aspects of Computing*. To appear.

[26] R. Giacobazzi and F. Ranzato. Uniform closures: order-theoretically reconstructing logic program semantics and abstract domain refinements. *Inform. and Comput.*, 145(2):153–190, 1998.

[27] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.

[28] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, 1980.

[29] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[30] M. F. Janowitz. Residuated closure operators. *Portug. Math.*, 26(2):221–252, 1967.

[31] Neil D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52(17(1)):307–339, 2004.

[32] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[33] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[34] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, J Cappert, and B Preneel. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, pages 527–533. CSREA Press, 2006.

[35] A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 70–81. ACM, 2007.

[36] I. Mastroeni and R. Giacobazzi. Weakening additivity in adjoining closures. *Order*. To appear.

[37] I. Mastroeni and D. Nikolic. Abstract program slicing: From theory towards an implementation. In *12th International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 452 – 467. Springer, 2010.

[38] Isabella Mastroeni and Damiano Zanardini. Abstract program slicing: an abstract interpretation-based approach to program slicing. *CoRR*, abs/1605.05104, 2016.

[39] 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.

[40] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 19:1–19:16. USENIX Association, 2007.

[41] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

Appendix: Technical results

The following result identifies the conditions on the relation between f and δ that guarantees that $\mathcal{R}^{\uparrow}_{\delta}(f)$ and $\mathcal{R}^{\downarrow}_{\delta}(f)$ do not trivially transform f, i.e., they do not transform f in the top $(\lambda x.\top)$ od the bottom $(\lambda x.x)$ of the functional domain.

Proposition 4 *Let $\langle L, \leq, \vee, \wedge, \top, \bot\rangle$ be a complete lattice, $f : L \to L$ and $\delta \in uco(L)$, we have that*

1. $\mathcal{R}^{\uparrow}_{\delta}(f) \neq \lambda x.\, \top$ *iff* $\forall x \in L.\, \delta(f(x)) \leq \delta(x)$ *iff* $\exists y \geq f(x).\, \delta(y) = \delta(x)$;
2. $\mathcal{R}^{\downarrow}_{\delta}(f) \neq \lambda x.\, \bot$ *iff* $\exists y \leq f(x).\, \delta(y) = \delta(x)$;

Proof:

1. By definition $\mathcal{R}^{\uparrow}_{\delta}(f) = \bigsqcap\{g \mid \forall x.\, \delta(x) = \delta(g(x)), f \sqsubseteq g\}$, therefore we can prove that

$$\begin{aligned}\mathcal{R}^{\uparrow}_{\delta}(f) \text{ exists } &\Leftrightarrow \{g \mid \forall x.\, \delta(x) = \delta(g(x)), f \sqsubseteq g\} \neq \emptyset\\ &\Leftrightarrow \forall x.\, \exists y.\, f(x) \leq y \ \wedge\ \delta(x) = \delta(y)\end{aligned}$$

($\Rightarrow$) Let $h \in \{g \mid \forall x.\, \delta(x) = \delta(g(x)), f \sqsubseteq g\}$ then, by definition, we have that $\forall x$ the element $h(x)$ is such that $f(x) \leq h(x)$ and $\delta(x) = \delta(h(x))$.

($\Leftarrow$) If $\forall x.\, \exists y.\, f(x) \leq y$ and $\delta(x) = \delta(y)$, then we can define function $h : L \to L$ that associates each element x with such y. Thus, we have that $h \in \{g \mid \forall x.\, \delta(x) = \delta(g(x)), f \sqsubseteq g\}$.

We now prove that:

$$\forall x.\exists y.\, f(x) \leq y \ \wedge\ \delta(x) = \delta(y) \ \Leftrightarrow\ \forall x. : \delta(f(x)) \leq \delta(x)$$

($\Rightarrow$) Assume that $\forall x.\exists y. f(x) \leq y \wedge \delta(x) = \delta(y)$, since δ is monotone we have that $\forall x.\delta(f(x)) \leq \delta(y) = \delta(x)$.

($\Leftarrow$) Assume that $\forall x.\, \delta(f(x)) \leq \delta(x)$, δ is extensive and therefore $f(x) \leq \delta(f(x))$ and since by hypothesis we have that $\delta(f(x)) \leq \delta(x)$, then $f(x) \leq \delta(x)$. Thus, $\forall x$ we can take $\delta(x)$ as element y such that $f(x) \leq y \ \wedge\ \delta(x) = \delta(y)$.

2. By definition $\mathcal{R}^{\downarrow}_{\delta}(f) = \bigsqcup\{g \mid \forall x.\, \delta(x) = \delta(g(x)), g \sqsubseteq f\}$, therefore we can prove that

$$\begin{aligned}\mathcal{R}^{\downarrow}_{\delta}(f) \text{ exists } &\Leftrightarrow \{g \mid \forall x.\, \delta(x) = \delta(g(x)), g \sqsubseteq f\} \neq \emptyset\\ &\Leftrightarrow \forall x.\, \exists y.\, f(x) \geq y \ \wedge\ \delta(x) = \delta(y)\end{aligned}$$

($\Rightarrow$) Let $h \in \{g \mid \forall x.\, \delta(x) = \delta(g(x)), f \sqsupseteq g\}$ then, by definition, we have that $\forall x$ the element $h(x)$ is such that $f(x) \geq h(x)$ and $\delta(x) = \delta(h(x))$.

($\Leftarrow$) If $\forall x.\, \exists y.\, f(x) \geq y$ and $\delta(x) = \varphi(y)$, then we can define function $h : L \to L$ that associates each element x with such y. Thus, we have that $h \in \{g \mid \forall x.\, \delta(x) = \delta(g(x)), g \sqsubseteq f\}$.

□

This means that we can find a non trivial simplification of function f revealing property δ, if and only if function f "loses" something of the property δ of the original element. Analogous for the refinement of f.

In the following we report the proofs of the results stated in the paper.

Theorem 1. Let L be a complete lattice, $f : L \to L$ and $\delta \in uco(L)$.

1. If δ is meet-uniform, then $\mathcal{R}^{\uparrow}_{\delta} \in uco(L \to L)$;
2. $\mathcal{R}^{\downarrow}_{\delta} \in lco(L \to L)$.

Proof:

(1) Let $H \overset{\text{def}}{=} \left\{ g : L \to L \,\middle|\, \forall x \in L : \delta(x) = \delta(g(x)), f \sqsubseteq g \right\}$. In order to prove that $\mathcal{R}^{\uparrow}_{\delta}(f) \in H$ we have to prove that $f \sqsubseteq \mathcal{R}^{\uparrow}_{\delta}(f)$ and that $\forall x \in L : \delta(x) = \delta(\mathcal{R}^{\uparrow}_{\delta}(f)(x))$. By definition, we have that $\mathcal{R}^{\uparrow}_{\delta}(f) = \lambda x. \bigwedge_{g \in H} g(x)$. Since every $g \in H$ is such that $\forall x \in L : f(x) \leq g(x)$ then $\forall x \in L : f(x) \leq \bigwedge_{g \in H} g(x)$ and therefore $f \sqsubseteq \mathcal{R}^{\uparrow}_{\delta}(f)$. Moreover, $\forall g \in H, x \in L$ we have that $\delta(x) = \delta(g(x))$ and since δ is meet-uniform for hypothesis we have that $\forall x \in L : \delta(\bigwedge_{g \in H} g(x)) = \delta(x)$ which means that $\forall x \in L : \delta(x) = \delta(\mathcal{R}^{\uparrow}_{\delta}(f)(x))$.

(2) Let $K \overset{\text{def}}{=} \left\{ g : L \to L \,\middle|\, \forall x \in L : \delta(x) = \delta(g(x)), g \sqsubseteq f \right\}$. In order to prove that $\mathcal{R}^{\downarrow}_{\delta} \in K$ we have to show that $\mathcal{R}^{\downarrow}_{\delta}(f) \sqsubseteq f$ and that $\forall x \in L : \delta(x) = \delta(\mathcal{R}^{\downarrow}_{\delta}(f)(x))$. By definition, we have that $\mathcal{R}^{\downarrow}_{\delta}(f) = \lambda x. \bigvee_{g \in H} g(x)$. Since every $g \in H$ is such that $\forall x \in L : g(x) \leq f(x)$ then $\forall x \in L : \bigvee_{g \in H} g(x) \leq f(x)$ and therefore $\mathcal{R}^{\downarrow}_{\delta}(f) \sqsubseteq f$. Moreover, $\forall g \in H, x \in L$ we have that $\delta(x) = \delta(g(x))$ and since δ is a closure it is join-uniform and therefore we have that $\forall x \in L : \delta(\bigvee_{g \in H} g(x)) = \delta(x)$ which means that $\forall x \in L : \delta(x) = \delta(\mathcal{R}^{\downarrow}_{\delta}(f)(x))$.

□

Theorem 2. Let L be a complete lattice, $f : L \to L$ and $\delta \in uco(L)$ such that the pair (f, δ) is not-$\mathcal{R}$-trivial. We have that:

1. If δ is meet-uniform then $\mathcal{R}^{\uparrow}_{\delta}(f) = \lambda x. \mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)$;
2. $\mathcal{R}^{\downarrow}_{\delta}(f) = \lambda x. \delta(x) \wedge f(x)$.

PROOF.

1. Let us prove the thesis in two steps. First of all we prove the equality

$$\mathcal{R}^{\uparrow}_{\delta}(f) = \lambda x. \bigwedge \left\{ y \in L \,\middle|\, f(x) \leq y, \delta(x) = \delta(y) \right\}$$

and in particular we prove that $\mathcal{R}^{\uparrow}_{\delta}(f)$ so characterized: (1) is greater than function f; (2) preserves the property δ; and (3) is the smallest function that satisfies point (1) and (2).

(a) $\forall x. f(x) \leq \bigwedge \left\{ y \in L \,\middle|\, f(x) \leq y, \delta(x) = \delta(y) \right\}$ since every y in the set is such that $f(x) \leq y$. This means that $f \sqsubseteq \mathcal{R}^{\uparrow}_{\delta}(f)$.

(b) Let us show that $\forall x.\ \delta(x) = \delta(\bigwedge\{\, y \in L \,|\, f(x) \leq Y, \delta(x) = \delta(y) \,\})$. Every y in this set is such that $\delta(x) = \delta(y)$, and since δ is meet-uniform we have the following equality $\delta(\bigwedge\{\, y \in L \,|\, f(x) \leq y, \delta(x) = \delta(y) \,\}) = \delta(x)$. This means that $\forall x.\ \delta(x) = \delta(\mathcal{R}^{\uparrow}_{\delta}(f)(x))$.

(c) We show that given a function $g : L \to L$ such that $f \sqsubseteq g$ and $\forall x \in L.\ \delta(x) = \delta(g(x))$ then $\mathcal{R}^{\uparrow}_{\delta}(f) \sqsubseteq g$, namely $\forall x \in L.\ \mathcal{R}^{\uparrow}_{\delta}(f)(x) \leq g(x)$. Observe that $\forall x \in L$ we have that $g(x) \in \{\, y \in L \,|\, f(x) \leq y, \delta(x) = \delta(y) \,\}$ and therefore $\bigwedge\{\, y \in L \,|\, f(x) \leq y, \delta(x) = \delta(y) \,\} \leq g(x)$ and therefore $\mathcal{R}^{\uparrow}_{\delta}(f)(x) \leq g(x)$.

Now let us prove that $\bigwedge\{\, y \in L \,|\, f(x) \leq y, \delta(x) = \delta(y) \,\} = \mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)$. Since δ is meet-uniform we have that

$$\delta(\bigwedge\{\, y \in L \,|\, f(x) \leq y, \delta(x) = \delta(y) \,\}) = \delta(x)$$

This means that we have to prove that for any given $x \in L$ the smallest element $w \in L$ such that $\delta(w) = \delta(x)$ and $f(x) \leq w$ is precisely $\mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)$.

Let us show that $\delta(\mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)) = \delta(x)$. Observe that, since δ is meet-uniform, $\delta(\mathtt{K}^{\wedge}_{\delta}(x)) = \delta(x)$. Moreover, observe that for any given element $w \in L$ such that $\mathtt{K}^{\wedge}_{\delta}(x) \leq w \leq \delta(x)$, then $\delta(w) = \delta(x)$. By hypothesis $w \leq \delta(x)$ and since δ is monotone and idempotent we have that $\delta(w) \leq \delta(x)$. Let $\delta(w) = m$, if $m \neq \delta(x)$, since $\mathtt{K}^{\wedge}_{\delta}(x) \leq w$ we would have that $\delta(\mathtt{K}^{\wedge}_{\delta}(x)) = m \neq \delta(x)$ that contradicts the hypothesis of δ being meet-uniform. We can easily observe that $\mathtt{K}^{\wedge}_{\delta}(x) \leq \mathtt{K}^{\wedge}_{\delta}(x) \vee f(x) \leq \delta(x)$, therefore we have that $\delta(\mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)) = \delta(x)$.

Let us now show that $\mathtt{K}^{\wedge}_{\delta}(x) \vee f(x)$ is the smallest one. Indeed, for every element $w \in L$ such that $\delta(w) = \delta(x)$ and $f(x) \leq w$ we have that $w \in \{y \mid \delta(x) = \delta(y)\}$ and therefore $\mathtt{K}^{\wedge}_{\delta}(x) = \bigwedge\{\, y \,|\, \delta(x) = \delta(y) \,\} \leq w$. $f(x) \leq w$ holds by hypothesis, and therefore $\mathtt{K}^{\wedge}_{\delta}(x) \vee f(x) \leq w$.

2. It follows by duality, by Proposition 4 and because $\mathtt{K}^{\vee}_{\delta}(x) = \delta(x)$.

□

Proposition 1. $\mathcal{R}^{\downarrow}_{\delta}(\lambda x.\top) = \delta$.

PROOF. By Def. 2 we have

$$\mathcal{R}^{\downarrow}_{\delta}(\lambda x.\top) = \bigsqcup\{\, g \,|\, \forall x \in L : \delta(x) = \delta(g(x)), g \sqsubseteq \lambda x.\top \,\}$$

Moreover, it trivially holds that $\delta(x) = \delta(\delta(x))$ and that $\delta \sqsubseteq \lambda x.\top$, which means that the property

$$\delta \in \{\, g : L \to L \,|\, \forall x \in L : \delta(x) = \delta(g(x)), g \sqsubseteq \lambda x.\top \,\}$$

For any function g in the set $\{\, g : L \to L \,|\, \forall x \in L : \delta(x) = \delta(g(x)), g \sqsubseteq \lambda x.\top \,\}$ and for every element $x \in L$ we have that $g(x) \leq \delta(g(x))$, since δ is extensive, and that $\delta(g(x)) = \delta(x)$ by hypothesis, which means that for every $x \in L.\ g(x) \leq \delta(x)$ namely that for any function g in the considered set $g \sqsubseteq \delta$.

This means that the property δ is the most abstract function that preserves δ itself, and this is implied by the fact that δ is extensive and idempotent. If we interpret this result on the domain of program semantics $L = \wp(\Sigma^*)$, we can say that the most abstract semantic transformation that preserves a given property δ is exactly the transformation that associates to each set of traces $X \in \wp(\Sigma^*)$ its fixpoint in δ, namely

$\delta(X)$. □

Theorem 3. Let L be a complete Boolean algebra, $\varphi \in uco(L)$ and $f : L \to L$ such that the pair (f, φ) is not-$\mathcal{R}$-trivial. We have that:

1. If φ is meet-uniform then $\mathcal{R}_\varphi^\uparrow \in uco(L \to L)$ is meet-uniform on f;
2. $\mathcal{R}_\varphi^\downarrow \in lco(L \to L)$ is join-uniform on f.

Proof:

1. By definition we have that $\mathcal{C}_\varphi^\downarrow(f)(x) = \bigwedge\{g(x) \mid \mathcal{R}_\varphi^\uparrow(g) = \mathcal{R}_\varphi^\uparrow(f)\}$. We have to prove that in general given $x \in L$ we have that $\mathcal{R}_\varphi^\uparrow(\mathcal{C}_\varphi^\downarrow(f))(x) = \mathcal{R}_\varphi^\uparrow(f)(x)$. From Theorem 2 this means that we have to prove that:

$$\mathtt{K}_\varphi^\wedge(x) \vee f(x) \;=\; \mathtt{K}_\varphi^\wedge(x) \vee \bigwedge \left\{ \, g(x) \,\middle|\, \forall z : \mathtt{K}_\varphi^\wedge(z) \vee f(z) = \mathtt{K}_\varphi^\wedge(z) \vee g(z) \, \right\}$$

The lattice L is a Boolean algebra, namely $\forall x \in L, Y \subseteq L$ it holds that $x \vee \bigwedge Y = \bigwedge\{x \vee y \mid y \in Y\}$ [28], thus we have:

$$\mathtt{K}_\varphi^\wedge(x) \vee \bigwedge \left\{ \, g(x) \,\middle|\, \forall z : \mathtt{K}_\varphi^\wedge(z) \vee f(z) = \mathtt{K}_\varphi^\wedge(z) \vee g(z) \, \right\} = \\ \bigwedge \left\{ \, \mathtt{K}_\varphi^\wedge(x) \vee g(x) \,\middle|\, \forall z : \mathtt{K}_\varphi^\wedge(z) \vee f(z) = \mathtt{K}_\varphi^\wedge(z) \vee g(z) \, \right\}$$

Since for each g in the set we have that $\mathtt{K}_\varphi^\wedge(x) \vee f(x) = \mathtt{K}_\varphi^\wedge(x) \vee g(x)$, then we have

$$\bigwedge \left\{ \, \mathtt{K}_\varphi^\wedge(x) \vee g(x) \,\middle|\, \forall z : \mathtt{K}_\varphi^\wedge(z) \vee f(z) = \mathtt{K}_\varphi^\wedge(z) \vee g(z) \, \right\} = \mathtt{K}_\varphi^\wedge(x) \vee f(x)$$

which concludes the proof.

2. By definition we have that $\mathcal{C}_\varphi^\uparrow(f)(x) = \bigvee \left\{ \, g(x) \,\middle|\, \mathcal{R}_\varphi^\downarrow(g) = \mathcal{R}_\varphi^\downarrow(f) \, \right\}$. We have to prove that in general given $x \in L$ we have that $\mathcal{R}_\varphi^\downarrow(\mathcal{C}_\varphi^\uparrow(f))(x) = \mathcal{R}_\varphi^\downarrow(f)(x)$. From Theorem 2 this means that we have to prove that:

$$\mathtt{K}_\varphi^\vee(x) \wedge f(x) \;=\; \mathtt{K}_\varphi^\vee(x) \wedge \bigvee \left\{ \, g(x) \,\middle|\, \forall z.\, \mathtt{K}_\varphi^\vee(z) \wedge f(z) = \mathtt{K}_\varphi^\vee(z) \wedge g(z) \, \right\}$$

The lattice L is a Boolean algebra, namely $\forall x \in L, Y \subseteq L$ it holds that $x \wedge \bigvee Y = \bigvee \left\{ \, x \wedge y \,\middle|\, y \in Y \, \right\}$ [28], thus we have:

$$\mathtt{K}_\varphi^\vee(x) \wedge \bigvee \left\{ \, g(x) \,\middle|\, \forall z.\, \mathtt{K}_\varphi^\vee(z) \wedge f(z) = \mathtt{K}_\varphi^\vee(z) \wedge g(z) \, \right\} = \\ \bigvee \left\{ \, \mathtt{K}_\varphi^\vee(x) \wedge g(x) \,\middle|\, \forall z.\, \mathtt{K}_\varphi^\vee(z) \wedge f(z) = \mathtt{K}_\varphi^\vee(z) \wedge g(z) \, \right\}$$

Since for each g in the set we have that $\mathtt{K}_\varphi^\vee(x) \wedge f(x) = \mathtt{K}_\varphi^\vee(x) \wedge g(x)$, then

$$\bigvee \left\{ \, \mathtt{K}_\varphi^\vee(x) \wedge g(x) \,\middle|\, \forall z.\, \mathtt{K}_\varphi^\vee(z) \wedge f(z) = \mathtt{K}_\varphi^\vee(z) \wedge g(z) \, \right\} = \mathtt{K}_\varphi^\vee(x) \wedge f(x)$$

which concludes the proof that $\mathtt{K}^{\vee}_{\varphi}(x) = \varphi(x)$.

□

Lemma 1 *Let L be a boolean algebra, $x, y, z \in L$, then we have that*

1. *If y is an atom then $y \leq x \vee z$ iff $y \leq x \;\vee\; y \leq z$;*
2. *If y is a coatom then $y \geq x \wedge z$ iff $y \geq x \;\wedge\; y \geq z$.*

Proof:

1. If $y \leq x \vee y \leq z$ trivially we have that $y \leq x \vee z$. Suppose now $y \leq x \vee z$, by definition this means that $y = y \wedge (x \vee z)$. By distributive law this is the same as $y = (y \wedge x) \vee (y \wedge z)$, which implies that $y \wedge x \leq y$ and $y \wedge z \leq y$. Being y an atom by hypothesis, this means that $y \wedge x \in \{\bot, y\}$ and $y \wedge z \in \{\bot, y\}$. Then if one of these two elements is y we have the thesis, otherwise both of them are $\bot$, which implies that their lub y is $\bot$, which is an absurd.
2. By duality.

□

Theorem 4 Let L be a complete Boolean algebra, $\varphi \in uco(L)$ and $f : L \to L$ such that the pair (f, φ) is not-$\mathcal{R}$-trivial. We have that:

1. If φ is meet-uniform then $\mathcal{C}^{\downarrow}_{\varphi}(f) = \lambda x.\ \bigvee \left\{ z \,\middle|\, z \leq f(x),\ z \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \right\}$;
2. $\mathcal{C}^{\uparrow}_{\varphi}(f) = \lambda x.\ \bigwedge \left\{ z \,\middle|\, z \geq f(x), z \vee \varphi(x) = \top \right\}$;

PROOF.

1. Note that by definition $\mathcal{C}^{\downarrow}_{\varphi}(f) = \bigwedge \left\{ z \,\middle|\, f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) \right\}$. We prove that $\forall y \in L$ we have $y \leq \bigwedge \left\{ z \,\middle|\, f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) \right\}$ iff we have $y \leq \bigvee \left\{ z \,\middle|\, z \leq f(x),\ z \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \right\}$. Note that this property can be proved considering only atoms since by induction on the distance of y from the bottom we can prove that $y \leq z$ iff $\forall \tilde{y} \leq y$, $\tilde{y}$ atom, $\tilde{y} \leq z$, being L a Boolean algebra and therefore generated by its atomic elements [28].

   Let $y \in L$ be an atom, then we have the following implications:

$$
\begin{aligned}
y \leq \bigvee \left\{ z \,\middle|\, z \leq f(x),\ z \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \right\}& \\
&\Rightarrow\ \exists\, z.\, y \leq z,\ z \leq f(x),\ z \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \text{ (By Lemma 1-1)} \\
&\Rightarrow\ y \leq f(x),\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) \leq z \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \\
&\Rightarrow\ y \leq f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x),\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \\
&\Rightarrow\ \forall z.\, (f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x)).\, y \leq z \vee \mathtt{K}^{\wedge}_{\varphi}(x),\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \\
&\Rightarrow\ \forall z.\, (f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x)).\, y \leq z \text{ (By Lemma 1-1)} \\
&\Rightarrow\ y \leq \bigwedge \left\{ z \,\middle|\, f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) \right\}
\end{aligned}
$$

   Now consider $y \in L$ atom such that $y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot$, then

$$
\begin{aligned}
y \leq \bigwedge \left\{ z \,\middle|\, f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) \right\},\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot& \\
&\Rightarrow\ \forall z.\, (f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x)).\, y \leq z,\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \\
&\Rightarrow\ \forall z.\, (f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x)).\, y \leq z \vee \mathtt{K}^{\wedge}_{\varphi}(x),\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \\
&\Rightarrow\ y \leq f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x),\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \\
&\Rightarrow\ y \leq f(x),\ y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \text{ (By Lemma 1-1)} \\
&\Rightarrow\ y \leq \bigvee \left\{ z \,\middle|\, z \leq f(x),\ z \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \right\}
\end{aligned}
$$

Now we have to prove that if $y \wedge \mathtt{K}^{\wedge}_{\varphi}(x) \neq \bot$, namely $y \leq \mathtt{K}^{\wedge}_{\varphi}(x)$ (being y an atom), then it cannot be $y \leq \bigwedge \{\, z \,|\, f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) \,\}$. Let us define $\overline{z} \overset{\text{def}}{=} \bigvee \{\, w \,|\, w \leq z,\ w \wedge \mathtt{K}^{\wedge}_{\varphi}(x) = \bot \,\}$, then we can note that $\overline{z} \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) = f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x)$. Hence if y atom is such that $y \leq \mathtt{K}^{\wedge}_{\varphi}(x)$ then $y \not\leq \overline{z}$ and therefore $y \not\leq \bigwedge \{\, z \,|\, f(x) \vee \mathtt{K}^{\wedge}_{\varphi}(x) = z \vee \mathtt{K}^{\wedge}_{\varphi}(x) \,\}$ since $\overline{z}$ is in this set.

2. By duality.

□

Corollary 1 Let D be a complete lattice, $L = \wp(D)$ and $\varphi \in uco(L)$. For each $f : L \leftarrow L$ such that the pair (f, φ) is not-$\mathfrak{R}$-trivial. We have that:

1. If φ is meet-uniform then $\mathfrak{C}^{\downarrow}_{\varphi}(f) = \lambda X.\, f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X)$;
2. $\mathfrak{C}^{\uparrow}_{\varphi}(f) = \lambda X.\, f(X) \cup (D \smallsetminus \varphi(X))$.

Proof:

1. Suppose $L = \wp(D)$, $X \in L$. We prove $\bigcup \{\, Z \,|\, Z \subseteq f(X),\ Z \cap \mathtt{K}^{\cap}_{\varphi}(x) = \varnothing \,\} = f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X)$. First of all note that $f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X) \subseteq f(X)$ and $(f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X)) \cap \mathtt{K}^{\cap}_{\varphi}(X) = \varnothing$. hence
$$\bigcup \{\, Z \,|\, Z \subseteq f(X),\ Z \cap \mathtt{K}^{\cap}_{\varphi}(x) = \varnothing \,\} \supseteq f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X)$$
We have to prove the other inclusion. In particular, if we can prove that for each $Z \in \{\, Z \,|\, Z \subseteq f(X),\ Z \cap \mathtt{K}^{\cap}_{\varphi}(x) = \varnothing \,\}$ we have $Z \subseteq (X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(X)$, then also the union is smaller. But this trivially holds since $Z \cap \mathtt{K}^{\cap}_{\varphi}(x) = \varnothing$ and $x \in Z$ imply $x \notin \mathtt{K}^{\cap}_{\varphi}(x)$, moreover $x \in Z$ implies $x \in f(X)$, hence $x \in Z$ implies $x \in f(X) \smallsetminus \mathtt{K}^{\cap}_{\varphi}(x)$.
2. Suppose $L = \wp(D)$, $X \in D$. We prove $\bigcap \{\, Z \,|\, Z \supseteq f(X), Z \cup \varphi(X) = D \,\} = f(X) \cup (D \smallsetminus \varphi(X))$.
First of all, note that $f(X) \cup (D \smallsetminus \varphi(X)) \supseteq f(X)$ and $f(X) \cup (D \smallsetminus \varphi(X)) \cup \varphi(X) = f(X) \cup D = D$, hence $f(X) \cup (D \smallsetminus \varphi(X)) \subseteq \bigcap \{\, Z \,|\, Z \supseteq f(X), Z \cup \varphi(X) = D \,\}$.
We have to prove the other inclusion. In particular, if we can prove that for each $Z \in \{\, Z \,|\, Z \supseteq f(X), Z \cup \varphi(X) = D \,\}$ we have $Z \supseteq f(X) \cup (L \smallsetminus \varphi(X))$, then also the intersection is greater. But this trivially holds since $Z \cup \varphi(X) = D$ implies that $Z \supseteq D \smallsetminus \varphi(X)$. Hence, if $Z \supseteq f(X)$ and $Z \cup \varphi(X) = D$, then $Z \supseteq f(X) \cup (D \smallsetminus \varphi(X))$.

□

Proposition 2. $\mathfrak{O}^{\delta}_{\varphi} = \lambda X.\, X \cup (\delta(X) \cap (\Sigma^* \smallsetminus \varphi(X)))$.

Proof:

$$\begin{aligned}\mathfrak{O}^{\delta}_{\varphi} &= \mathfrak{R}^{\downarrow}_{\delta}(\mathfrak{C}^{\uparrow}_{\varphi}(id)) = \mathfrak{R}^{\downarrow}_{\delta}(\lambda X.\, X \cup (\Sigma^* \smallsetminus \varphi(X)))\\ &= \lambda X.\, \delta(X) \cap (X \cup (\Sigma^* \smallsetminus \varphi(X)))\\ &= \lambda X.\, X \cup (\delta(X) \cap (\Sigma^* \smallsetminus \varphi(X)))\end{aligned}$$

where the last equality holds by the distributivity law and by extensivity of δ.

□

Corollary 2. $\mathfrak{O}^{\delta}_{\varphi}(X) \neq X$ if and only if $\delta(X) \not\subseteq \varphi(X)$.

Proof: By Definition 5 and Proposition 2 we have that $\mathfrak{O}^{\delta}_{\varphi}(X) \neq X$ if and only if $\delta(X) \cap (\Sigma^* \smallsetminus \varphi(X)) \neq \emptyset$ and this holds if and only if $\delta(X) \not\subseteq \varphi(X)$.

□

Corollary 3. A weakened property-driven obfuscator $\hat{\mathfrak{O}}^{\delta}_{\varphi}(X)$ exists if and only if

$$\exists Y \in \wp(\Sigma^*).\, \delta(Y) \not\subseteq \varphi(Y).$$

Proof: By definition of weakened property-driven obfuscator we have that $\hat{\mathfrak{O}}^{\delta}_{\varphi}(X)$ exists if and only if $\mathtt{K}_{\delta}(X) \cap (\wp(\Sigma^*) \smallsetminus \mathtt{K}_{\varphi}(X)) \neq \varnothing$. Clearly this holds if and only if $\mathtt{K}_{\delta}(X) \not\subseteq \mathtt{K}_{\varphi}(X)$ if and only if $\delta \not\sqsubseteq \varphi$ if and only if $\exists Y \in \wp(\Sigma^*).\, \delta(Y) \not\subseteq \varphi(Y)$.

□

Cover Letter

The paper to which this letter is attached is a revision of paper 14-672

- first submission September 2014
- first revision submission 14-672-R march 2015
- second revision submission july 2016 14-672-RR

While preparing the revised version, we have followed the suggestions of the reviewers and we have implemented all the comments.

REVIEWER 1 p.3: It is not at all clear which property you're concealing in the parity example: do you want to conceal the parity of the intermediate values of s?

Yes, we have better explained that we are interested in hiding the values assumed by variable s during computation.

p.12: Can you give some intuition why a property is modeled by a uco, as opposed, say, to a set of executions, as is common in many frameworks? Is it because while safety and liveness properties correspond to sets of executions, more general properties need not be?

at the top of page 13 (of the new version) we have added the following sentence that explains the fact that an uco is indeed the map associating each semantics with its property expressed as set of traces of execution.

"In other words, the attacker is the map (uco) associating with each semantics S (set of executions) the smallest set of executions containing S and satisfying a corresponding invariant (the S observable property)."

p.16: Why do we care about both revelation from above and from below? Which best applies when?

Since there are two possible ways to minimally transform a function in order to make it reveal a property (from above and from below) we decided to consider both of them for completeness of the framework.

p.22: I don't understand the sentences after Corollary 2. I think the point is that you only need to add the abstract semantics and not the whole original semantics in order to conceal phi and reveal delta. It's probably worth rewriting to make this clearer. (Assuming I understood correctly. If I did not, then you definitely need to rewrite!)

you understood correctly we tried to make it more clear in the text

p.24: How does section 7.1 relate to anything in the rest of the article? I understand it is discussing the case where you want to both reveal the exact original semantics and also conceal something, but you never talk in terms of revealed/concealed properties, and the technical details seem to be independent of sections 4 and 5. Which is presumably fine, but it should be explained.

The section 7.1 describes a systematic technique for building an obfuscator protecting/concealing a given property, when the property to reveal is simply the I/O behaviour. We have underlined this at the beginning of the second paragraph of the section.

REVIEWER 2 All the suggested comments have been implemented

REVIEWER 3 The new examples in the paper have improved the motivation and the clarity of exposition is improved. Nice job. I think the significance of the work could still be motivated better for the reader who lacks familiarity with abstract interpretation or obfuscation techniques; I fear that even readers who are unfamiliar with just one of abstract interpretation and obfuscation will still need a stronger justification for digging deeply into this paper. It would be great to pull up earlier in the paper an example that more clearly suggests the value of the formal tools developed in the paper.

at end of page 4 (in the revised manuscript) we have continued the example given earlier in the introduction in order to show the intuition of the characterization of the maximal obfuscator provided by the proposed strategy