Hiding Information in Completeness Holes

New perspectives in code obfuscation and watermarking

Roberto Giacobazzi
Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, 37134 Verona, Italy
E-mail: roberto.giacobazzi@univr.it

Abstract

In this paper we show how abstract interpretation, and more specifically completeness, provides an adequate model for reasoning about code obfuscation and watermarking. The idea is that making a program obscure, or equivalently hiding information in it, corresponds to force an interpreter (the attacker) to become incomplete in its attempts to extract information about the program. Here abstract interpretation provides the model of the attacker (malicious host) and abstract interpretation transformers provide driving methods for understanding and designing new obfuscation and watermarking strategies: Obfuscation corresponds to make the malicious host incomplete and watermarking corresponds to hide secrets where incomplete attackers cannot extract them unless some secret key is given.

1. Introduction

Protection via obscurity is gaining more and more attention as a practical method for DRM and IPP in software design. Malicious host attacks exploit sensitive information leakage e.g., by source code analysis, knowledge extraction by static and dynamic analysis, program decomposition for code reuse, source code disassembly and decompilation for reverse engineering, and integrity corruption for code haking [36]. Negative results on the impossibility of perfect and universal obscurity, such as [4], did not dishearten researchers in developing methods and algorithms for hiding sensitive information in programs. As well as Rice's theorem represented the greatest challenge for the development of automatic program analysis and verification tools, the impossibility of obfuscation against malicious host attacks is a major challenge for developing concrete techniques which are sufficiently robust that an attacker is in trouble for a sufficient amount of time in trying to defeat them. Code obfuscation, software watermarking and steganography are successful examples of these tools, gaining an increasing importance in the quality of critical software products [11].

Hiding information means both hiding as making it imperceptible and obscuring as making it incomprehensible [43]. In programming, perception and comprehension of code's structure and behaviour are deep semantic concepts, which depend on the relative degree of abstraction of the observer, which corresponds precisely to program semantics. In this paper we show that abstract interpretation can be used as an adequate model for developing a unifying theory of information hiding in software, by modeling observers (i.e., malicious host attackers) $\mathcal{O}$ as suitable abstract interpreters. An observation can be any static or dynamic interpretation of programs intended to extract properties from its semantics and abstract interpretation [14] provides the best framework to understand semantics at different levels of abstraction. The long standing experience in digital media protection by obscurity is inspiring here. It is known that practical steganography is an issue where compression methods are inefficient: *"Where efficient compression is available, information hiding becomes vacuous."* [3]. This means that the gain provided by compression can be used for hiding information. This, in contrast to cryptography, strongly relies upon the understanding of the supporting media: if we have a source which is completely understandable, i.e., it can be perfectly compressed, then steganography becomes trivial. In programming languages, a complete understanding of semantics means that no loss of precision is introduced by approximating data and control components while analysing computations. Complete abstractions [15, 34] model precisely the complete understanding of program semantics by an approximate observer, which corresponds to the possibility of replacing, with no loss of precision, concrete computations with abstract ones —some sort of perfect semantic compressibility around a given property. This includes, for instance, both static and dynamic, via monitoring, approaches to information disclosure and reverse engineering [18]. The lack of completeness of the observer is therefore the corresponding of its poor understanding of program semantics, and provides the key aspect for understanding and designing a new family of methods and tools for software steganography and obfuscation. Consider the simple statement, $C : \mathtt{x} = \mathtt{a} * \mathtt{b}$, multiplying $\mathtt{a}$ and $\mathtt{b}$, and storing the result in $\mathtt{x}$. An automated program sign analysis replacing concrete computations with approximated ones (i.e., the rule of signs) is able to catch, with no loss of precision, the intended sign behaviour of C because the sign abstraction $\mathcal{O} = \{+, 0, -\}$, is complete for integer multiplication. If we replace C with $\mathfrak{O}(C)$: $\mathtt{x} = 0$; $\mathtt{if\ b} \leq 0\ \mathtt{then}\ \{\mathtt{a} = -\mathtt{a}; \mathtt{b} = -\mathtt{b}\}$; $\mathtt{while\ b} \neq 0\ \{\mathtt{x} = \mathtt{a} + \mathtt{x}; \mathtt{b} = \mathtt{b} - 1\}$ we obfuscate the observer $\mathcal{O}$ because the rule of signs is incomplete for integer addition. Intervals, i.e., a far more concrete observer, are required in order to automatically understand the sign computed in $\mathfrak{O}(C)$. We show how this idea can be extended to arbitrary obfuscation methods and exploited for code steganography, providing the basis for a unifying theory for these technologies in terms of abstract interpretation. By some examples and ideas, we show how obfuscation can be viewed as a program transformation making abstractions incomplete and at the same time we show how watermark extraction can be viewed as a complete abstract interpretation against a secret program property, extending abstract watermarking [19] to any watermarking method. Both obfuscation and watermarking can be specified as transformers to achieve completeness/incompleteness in abstract interpretation [29], provided that the transformed code does not interfere with the expected input/output behaviour of programs. This latter correctness criteria can be again specified as a completeness problem by considering abstract non-interference [27] as the method for controlling information leakage in obfuscation and steganography. Our approach is language independent and can be applied to most known obfuscation and watermarking methods, providing a common ground for their understanding and comparison.

2. Basic mathematical notation

If S and T are sets, then $\wp(S)$ denotes the powerset of S and $S \times T$ denotes the Cartesian product of S and T, If $f : S \longrightarrow T$, $Y \subseteq S$, and $X \subseteq T$ then $f(Y) \overset{\text{def}}{=} \{f(y) \mid y \in Y\}$ and $f^{-1}(X) \overset{\text{def}}{=} \left\{ x \mid f(x) \in X \right\}$. We will often denote $f(\{x\})$ as $f(x)$ and use lambda notation for functions. $f \circ g \overset{\text{def}}{=} \lambda x.\ f(g(x))$. $\langle C, \leq \rangle$ denotes a poset C with ordering relation $\leq$, while $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice C, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, top and bottom element $\top$ and $\bot$ respectively. $id \overset{\text{def}}{=} \lambda x.\ x$ and $\mathbb{T} \overset{\text{def}}{=} \lambda x.\ \top$. The point-wise ordered set of monotone functions, denoted $C_1 \xrightarrow{\text{m}} C_2$, is a complete lattice $\langle C_1 \xrightarrow{\text{m}} C_2, \sqsubseteq, \sqcup, \sqcap, \mathbb{T}, \lambda x.\ \bot \rangle$. $f : C_1 \longrightarrow C_2$ is (completely) additive if f preserves *lub*'s of all subsets of C_1 (emptyset included). Continuity, denoted $\xrightarrow{\text{c}}$, holds when f preserved *lubs*'s of chains. Co-additivity and co-continuity are dually defined. Weaker notions of additivity have been studied in the context of abstract domain transformers. $f : C_1 \longrightarrow C_2$ is *join-uniform* [32] if for all $Y \subseteq C_1$, $(\exists \bar{x} \in Y.\ \forall y \in Y.\ f(y) = f(\bar{x})) \Rightarrow (\exists \bar{x} \in Y.\ f(\bigvee Y) = f(\bar{x}))$. Meet-uniformity is defined dually.

3. Abstract domains

Abstract interpretation is a general theory for deriving sound approximations of the semantics of discrete dynamic systems, e.g., programming languages [14]. We consider Galois connection-based abstract interpretation [15]. $\alpha : C \xrightarrow{\text{m}} A$ and $\gamma : A \xrightarrow{\text{m}} C$ form an *adjunction* or a *Galois connection* (GC), denoted $\langle C, \alpha, \gamma, A \rangle$, if $\forall x \in C, \forall y \in A$: $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. α (resp. γ) is the *left-* (*right-*) *adjoint* to γ (α) and it is an additive (co-additive) function. Additive and co-additive functions f admit respectively right and left adjoint: $f^+ \overset{\text{def}}{=} \lambda x.\ \bigvee \left\{ y \mid f(y) \leq x \right\}$ and $f^- \overset{\text{def}}{=} \lambda x.\ \bigwedge \left\{ y \mid x \leq f(y) \right\}$ respectively. Remember that $(f^+)^- = (f^-)^+ = f$ (see [5]). If $\forall a \in A$: $\alpha(\gamma(a)) = a$, then $\langle C, \alpha, \gamma, A \rangle$ is a Galois insertion (GI). In GC-based abstract interpretation the concrete and abstract domains, C and A, are complete lattices [14]. An *upper (lower) closure* $\rho : C \xrightarrow{\text{m}} C$ in *uco*(C) (*lco*(C)) is any idempotent and extensive: $\forall x \in C.\ x \leq \rho(x)$ (reductive: $\forall x \in C.\ x \geq \rho(x)$) operator. Closure operators are uniquely determined by the set of their fix-points $\rho(C)$. $X \subseteq C$ is the set of fix-points of $\rho \in$ *uco*(C) iff X is a *Moore-family* of C, i.e., $X = \mathcal{M}(X) \overset{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$, $\wedge\varnothing = \top \in \mathcal{M}(X)$, iff X is isomorphic to an abstract domain A in a GI $\langle C, \alpha, \gamma, A \rangle$, i.e., $A \cong \rho(C)$ with $\iota : \rho(C) \longrightarrow A$ and $\iota^{-1} : A \longrightarrow \rho(C)$

being an isomorphism, and $\langle C, \iota\circ\rho, \iota^{-1}, A\rangle$ is the GI, i.e., $\rho = \gamma\circ\alpha$. Dual properties can be derived for lower closures. $uco(C)$ is therefore isomorphic to the so called *lattice of abstract interpretations of* C [15]. If C is a complete lattice then $uco(C)$ and $lco(C)$ ordered point-wise are also complete lattices. For upper closures $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \mathbb{T}, id\rangle$ where for every $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i\in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$; and $(\sqcup_{i\in I}\rho_i)(x) = x \Leftrightarrow \forall i \in I.\ \rho_i(x) = x$. Dual properties can be derived for $\langle lco(C), \sqsubseteq, \sqcup, \sqcap, id, \lambda x.\ \bot\rangle$. In the following we will find particularly convenient to identify closure operators (and therefore abstract domains) with their sets of fix-points. Let $\rho \in uco(C)$, its *disjunctive completion* is $\curlyvee(\rho) = \sqcup\{\eta \in uco(C) | \eta \sqsubseteq \rho$ and η is additive$\}$. ρ is disjunctive iff $\rho(C)$ is a complete sublattice of C iff $\curlyvee(\rho) = \rho$ (cf. [15]). If π is a partition (viz. an equivalence relation), then $[\cdot]_\pi$ is the corresponding equivalence class. A closure $\eta \in uco(\wp(C))$ induces a partition on C: $\{\ [x]_\eta \mid x \in C\ \}$, where $[x]_\eta \stackrel{\text{def}}{=} \{\ y \mid \eta(x) = \eta(y)\ \}$. The most concrete closure that induces the same partition of values as η is $\Pi(\eta) \stackrel{\text{def}}{=} \curlyvee\left(\{\ [x]_\eta \mid x \in C\ \}\right)$. η is *partitioning* if $\eta = \Pi(\eta)$ [45]. *Reduced product* and *power* are the best known operations to compose abstract domains in order to exploit respectively the attribute independent and relational properties of programs [15]. The reduced product of a family of domains $\{\rho_i\}_{i\in I} \subseteq uco(C)$ is $\sqcap_{i\in I}\rho_i$. The reduced relative power in [33] is a generalization over arbitrary quantales of Cousot's original reduced power [15]. Let $\langle C, \leq, \bowtie\rangle$ be a *semi-quantale* [48], i.e. an algebraic structure where $\langle C, \leq\rangle$ is a complete lattice and $\bowtie$: $C \times C \longrightarrow C$ is an associative, commutative, and additive binary operation. Given a pair of Galois connections between a concrete domain C and two domains A_1 and A_2: $\langle C, \alpha_1, \gamma_1, A_1\rangle$ and $\langle C, \alpha_2, \gamma_2, A_2\rangle$, we define the (relative) reduced power of A_1 and A_2, as the set $A_1 \xrightarrow{\bowtie} D_2 \subseteq A_1 \xrightarrow{\text{m}} A_2$ of all the monotone functions defined as $\lambda x.\ \alpha_2(c \bowtie \gamma_1(x))$ with $c \in C$. We have that $\langle C, \alpha, \gamma, A_1 \xrightarrow{\bowtie} A_2\rangle$ with $\alpha \stackrel{\text{def}}{=} \lambda c.\ \lambda x.\ \alpha_2(c \bowtie \gamma_1(x))$.

4. The programming language

We consider a simple C-like non-deterministic imperative language, where programs in $\mathbb{P}$ are commands over standard expressions e, evaluated in the set of values $\mathbb{V}$:

$$\begin{array}{rcl} c & ::= & \textbf{nil} \mid x = e \mid c\ ;\ c \mid c \,\Box\, c \mid \textbf{return}(e) \mid \\ & & \textbf{if}\ e\ \{c\}\ \{c\}\ \mid \textbf{while}\ e\ \{c\} \end{array}$$

$Var(P)$ will denote the set of variables of P. The operational semantics is standard [51] and naturally induces a transition relation on the set of states Σ, denoted $\rightsquigarrow$, specifying the relation between a state and its possible successors in a transition system $\langle \Sigma, \rightsquigarrow\rangle$. For the sake of simplicity we consider only finite (terminating) computations in Σ^*. The empty sequence is denoted ε. The length of $\sigma \in \Sigma^*$ is denoted $|\sigma| \in \mathbb{N}$ and its i-th element is denoted σ_i. A non-empty finite *trace* $\sigma \in \Sigma^*$ is a finite sequence of consecutive states such that for all $i < |\sigma|$: $\sigma_i \rightsquigarrow \sigma_{i+1}$. The *maximal finite trace semantics* [16] of a transition system associated with a program P is denoted $(\!|P|\!)$, where $(\!|P|\!)^n = \{\sigma \in \Sigma^* \| \sigma| = n, \forall i \in [1, n)\ .\ \sigma_{i-1} \rightsquigarrow \sigma_i\}$ and, if $T \subseteq \Sigma$ is the set of final/blocking states, then $(\!|P|\!) = \bigcup_{n>0}\{\sigma \in (\!|P|\!)^n | \sigma_{n-1} \in T\}$. If $\sigma \in (\!|P|\!)$, then $\sigma_\dashv$ and $\sigma_\vdash = \sigma_0$ denote, respectively, the final and initial state of σ. The semantics $(\!|P|\!)$ has been obtained in [16] as a fix-point of the monotone operator $F_P : \wp(\Sigma^*) \xrightarrow{\text{m}} \wp(\Sigma^*)$ defined on traces as $F_P(X) \stackrel{\text{def}}{=} \Sigma \cup X \frown (\!|P|\!)^2$, where $\frown$ is sequence concatenation. In this case: $(\!|P|\!) = lfp^{\subseteq}_{\varnothing} F_P \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} F_P^n(\varnothing)$ [16]. The *(angelic) denotational semantics* associates (forward) input/output functions with programs, by ignoring non-termination. This semantics is derived in [13] by abstract interpretation from the maximal trace semantics with abstraction $\mathcal{D}(X) \stackrel{\text{def}}{=} \lambda s \in \Sigma.\ \{\sigma_\dashv | \sigma \in X \wedge s = \sigma_\vdash\}$, such that $\langle\langle\wp(\Sigma^*), \subseteq\rangle, \mathcal{D}, \mathcal{D}^+, \langle\Sigma \longrightarrow \wp(\Sigma), \sqsubseteq\rangle\rangle$ is a GI. It is well known that a function $[\![P]\!]$ can be associated with each $P \in \mathbb{P}$, inductively on its syntax, such that $[\![P]\!] \stackrel{\text{def}}{=} \mathcal{D}((\!|P|\!))$. The *weakest liberal precondition* semantics is instead defined as $\textbf{wlp}[\![P]\!] \stackrel{\text{def}}{=} \mathcal{W}((\!|P|\!))$ where $\mathcal{W}(X) \stackrel{\text{def}}{=} \lambda s \in \Sigma.\ \{\sigma_\vdash | \sigma \in X \wedge s = \sigma_\dashv\}$, such that $\langle\wp(\Sigma^\infty), \subseteq\rangle, \mathcal{W}, \mathcal{W}^+, \langle\langle\Sigma \longrightarrow \wp(\Sigma), \sqsubseteq\rangle\rangle$ is a GI. Note that, when lifted to sets of states, $[\![\cdot]\!]$ and $\textbf{wlp}[\![\cdot]\!]$ are adjoint functions, i.e., $[\![P]\!]^+ = \textbf{wlp}[\![P]\!]$.

5. Soundness

There are two equivalent ways to express the soundness of an abstraction [15]. Let $f : C \xrightarrow{\text{m}} C$, $\langle C, \alpha, \gamma, A\rangle$ be a GI, and $f^\sharp : A \xrightarrow{\text{m}} A$. Then $\langle C, \alpha, \gamma, A\rangle$ and $f^\sharp$ provide a sound abstraction of f if $\alpha\circ f \leq f^\sharp\circ\alpha$, or equivalently (by adjunction) if $f\circ\gamma \leq \gamma\circ f^\sharp$. The *best correct approximation* of f is $f^{bca} \stackrel{\text{def}}{=} \alpha\circ f\circ\gamma$ (or equivalently $\gamma\circ\alpha\circ f\circ\gamma\circ\alpha$). It is known that $f^\sharp$ is sound iff $f^{bca} \sqsubseteq f^\sharp$ and this implies that $\alpha(lfp(f)) \leq lfp(f^{bca}) \leq lfp(f^\sharp)$ [15]. In the following, if $[\![P]\!]$ is specified as fixpoint of (a combination of) predicate-transformers $F_P : C \xrightarrow{\text{c}} C$, and $\rho \in uco(C)$, we denote by $[\![P]\!]^\rho$ the (fixpoint) semantics associated with $F_P^{bca} = \rho\circ F_P\circ\rho$. $[\![P]\!]^\rho$ is the best correct abstract interpretation of P in ρ. In this case $\rho([\![P]\!]) \leq [\![P]\!]^\rho$.

6. Completeness

While the above definitions of soundness are equivalent, they are not equivalent when equality is required, i.e., when we consider completeness [15, 34, 30]. Even if both imply that $\gamma\circ\alpha(\llbracket P\rrbracket) = \llbracket P\rrbracket^{\gamma\circ\alpha}$, $\alpha\circ f = f^\sharp\circ\alpha$ means that no loss of precision is accumulated by approximating the input arguments of a given semantic function while $f\circ\gamma = \gamma\circ f^\sharp$ means that no loss of precision is accumulated by approximating the result of computations on abstract objects. We will follow [30] where the first is called *backward* ($\mathcal{B}$) and the second is called *forward* ($\mathcal{F}$) completeness. The key point in this construction is that there exists an either $\mathcal{B}$ or $\mathcal{F}$-complete abstract function $f^\sharp$ in an abstract domain $\rho \in \mathit{uco}(C)$ iff the best correct approximation $\rho\circ f\circ\rho$ of f is respectively either $\mathcal{B}$ or $\mathcal{F}$ complete [34], respectively $\rho\circ f = \rho\circ f\circ\rho$ or $f\circ\rho = \rho\circ f\circ\rho$. This means that both $\mathcal{F}$ and $\mathcal{B}$ completeness are properties of the underlying abstract domain and of the concrete function f. Therefore, by definition, completeness can be achieved either by transforming abstract domains or by transforming functions, which are in our case semantics.

6.1 Domain completeness

The problem of making abstract domains $\mathcal{B}$-complete has been solved in [34]. These results have been extended to $\mathcal{F}$-completeness in [30]. Let $f : C \xrightarrow{c} C$ and $\rho, \eta \in \mathit{uco}(C)$. $\langle\rho,\eta\rangle$ is a pair of $\mathcal{B}(\mathcal{F})$-complete abstractions for f if $\rho\circ f = \rho\circ f\circ\eta$ ($f\circ\eta = \rho\circ f\circ\eta$). In the following we denote by $\mathcal{F}(C,f) \overset{\text{def}}{=} \left\{ \langle\rho,\eta\rangle \,\middle|\, f\circ\eta = \rho\circ f\circ\eta \right\}$ and $\mathcal{B}(C,f) \overset{\text{def}}{=} \left\{ \langle\rho,\eta\rangle \,\middle|\, \rho\circ f = \rho\circ f\circ\eta \right\}$. A pair of domain transformers can be associated with any completeness problem. We follow [26, 31] by defining a *domain refinement* and *simplification* as any function $\tau : \mathit{uco}(C) \xrightarrow{m} \mathit{uco}(C)$ such that $X \subseteq \tau(X)$ and $\tau(X) \subseteq X$ respectively. In [34] and [30], a constructive characterization of the most abstract refinement, called *complete shell*, and of the most concrete simplification, called *complete core*, of any domain, making it $\mathcal{F}$ or $\mathcal{B}$ complete, for a given continuous function f, is given as a solution of a simple domain equation. Consider the following basic operators on closures:

$$\boxed{\begin{array}{l}
R_f^{\mathcal{F}} \overset{\text{def}}{=} \lambda X.\, \mathcal{M}(f(X)) \\
R_f^{\mathcal{B}} \overset{\text{def}}{=} \lambda X.\, \mathcal{M}(\bigcup_{y\in X} \max(f^{-1}(\downarrow y))) \\
C_f^{\mathcal{F}} \overset{\text{def}}{=} \lambda X.\, \left\{ y \in C \,\middle|\, f(y) \subseteq X \right\} \\
C_f^{\mathcal{B}} \overset{\text{def}}{=} \lambda X.\, \left\{ y \in C \,\middle|\, \max(f^{-1}(\downarrow y)) \subseteq X \right\}
\end{array}}$$

Let $\ell \in \{\mathcal{F},\mathcal{B}\}$. In [34] the authors proved that the only interesting cases, as far as the refinement and simplification towards ℓ-completeness are concerned, are respectively the most concrete $\beta \sqsupseteq \rho$ such that $\langle\beta,\eta\rangle$ is ℓ-complete and the most abstract $\beta \sqsubseteq \eta$ such that $\langle\rho,\beta\rangle$ is ℓ-complete. The ℓ-complete shell of η is $\mathcal{R}_f^{\ell,\rho}(\eta) \overset{\text{def}}{=} \eta \sqcap R_f^\ell(\rho)$ and the ℓ-complete core of ρ is $\mathcal{C}_f^{\ell,\eta}(\rho) \overset{\text{def}}{=} \rho \sqcup C_f^\ell(\eta)$. Note that, when f is additive $\max\left\{ x \,\middle|\, f(x) \le y \right\} = \bigvee\left\{ x \,\middle|\, f(x) \le y \right\} = f^+$, and therefore $\mathcal{B}(C,f) = \mathcal{F}(C,f^+)$ (cf. [30]). Clearly, when we consider $f : C \xrightarrow{c} C$ and the constraint $\eta = \rho$, the above construction requires a fixpoint iteration on abstract domains: $\mathcal{R}_f^\ell(\rho) = \mathit{gfp}(\lambda X.\, \rho \sqcap R_f^\ell(X))$ and $\mathcal{C}_f^\ell(\rho) = \mathit{lfp}(\lambda X.\, \rho \sqcup C_f^\ell(X))$ are called respectively the *absolute ℓ-complete shell* and *core* of ρ for f. Note that $\mathcal{R}_f^\ell \in \mathit{lco}(\mathit{uco}(C))$ and $\mathcal{C}_f^\ell \in \mathit{uco}(\mathit{uco}(C))$ (see [34]). It is worth noting that ℓ-complete cores and shells are adjoint abstract domain transformers, i.e., for any $\rho, \eta \in \mathit{uco}(C)$: $C_f^\ell(\eta) \sqsubseteq \rho \Leftrightarrow \eta \sqsubseteq R_f^\ell(\rho)$, which, by definition, implies that $\mathcal{C}_f^{\ell,\eta}(\rho) \sqsubseteq \rho \Leftrightarrow \eta \sqsubseteq \mathcal{R}_f^{\ell,\rho}(\eta)$.

6.2 Semantic completeness

Because in general $\mathcal{B}(C,f) = \mathcal{F}(C,f^+)$, then we have $\mathcal{B}(C,\llbracket P\rrbracket) = \mathcal{F}(C,\mathbf{wlp}\llbracket P\rrbracket)$. Being $\mathcal{F}$-completeness usually simpler to handle (cf. [30]), in the following of this section we consider $\mathcal{F}$-completeness only. The problem of minimally transforming semantics in order to achieve ($\mathcal{F}$-) completeness has been firstly addressed and solved in [29]. The authors proved that the set $\{f : C \xrightarrow{m} C \mid \rho\circ f\circ\eta = f\circ\eta\}$ is an upper closure operator of $\langle C \xrightarrow{m} C, \sqsubseteq\rangle$, and it is a lower closure iff ρ is additive. This means that there exist the closest complete approximations from above and from below of any given (possibly incomplete) semantics. For any $f \in C \xrightarrow{m} C$ and $\eta, \rho \in \mathit{uco}(C)$ define:

$$\mathbb{F}^{\uparrow}_{\eta,\rho} \overset{\text{def}}{=} \lambda f.\lambda x. \begin{cases} \rho\circ f(x) & \text{if } x \in \eta(C) \\ f(x) & \text{otherwise} \end{cases}$$

$$\mathbb{F}^{\downarrow}_{\eta,\rho} \overset{\text{def}}{=} \lambda f.\lambda x. \begin{cases} \rho^+\circ f(x) & \text{if } x \in \eta(C) \\ f(x) & \text{otherwise} \end{cases}$$

If $f : C \xrightarrow{m} C$, then

$$\mathbb{F}^{\uparrow}_{\eta,\rho}(f) = \bigsqcap\left\{ h : C \longrightarrow C \,\middle|\, f \sqsubseteq h,\ \rho\circ h\circ\eta = h\circ\eta \right\}$$

$$\mathbb{F}^{\downarrow}_{\eta,\rho}(f) = \bigsqcup\left\{ h : C \longrightarrow C \,\middle|\, f \sqsupseteq h,\ \rho\circ h\circ\eta = h\circ\eta \right\}$$

moreover, $\mathbb{F}^{\uparrow}_{\eta,\rho}(f)$ and $\mathbb{F}^{\downarrow}_{\eta,\rho}(f)$ are both $\mathcal{F}$-complete and if $\rho \in \mathit{uco}(C)$ is additive then $(\mathbb{F}^{\uparrow}_{\eta,\rho})^+ = \mathbb{F}^{\downarrow}_{\eta,\rho}$.

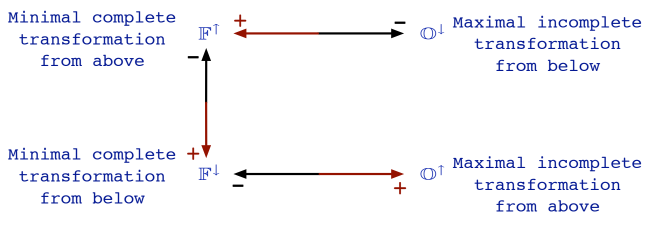
Minimal complete transformation from above
$\mathbb{F}^{\uparrow}$
+
−
−
$\mathbb{O}^{\downarrow}$
Maximal incomplete transformation from below
Minimal complete transformation from below
+
$\mathbb{F}^{\downarrow}$
−
+
$\mathbb{O}^{\uparrow}$
Maximal incomplete transformation from above

Figure 1. Basic semantic transformers.

One of the major achievements in [29] is the construction of a symmetric family of semantics transformers which induce maximal incompleteness. These are the adjoint operations associated with $\mathbb{F}^{\downarrow}$ and $\mathbb{F}^{\uparrow}$, see Figure 1:

$$\mathbb{O}^{\uparrow}_{\eta,\rho}(f) \overset{\text{def}}{=} \bigsqcup \left\{ g : C \longrightarrow C \mid \mathbb{F}^{\downarrow}_{\eta,\rho}(g) = \mathbb{F}^{\downarrow}_{\eta,\rho}(f) \right\}$$
$$\mathbb{O}^{\downarrow}_{\eta,\rho}(f) \overset{\text{def}}{=} \bigsqcap \left\{ g : C \longrightarrow C \mid \mathbb{F}^{\uparrow}_{\eta,\rho}(g) = \mathbb{F}^{\uparrow}_{\eta,\rho}(f) \right\}$$

$\mathbb{O}^{\uparrow}_{\eta,\rho}(f) \in \mathit{uco}(C \longrightarrow C)$ iff ρ is additive and ρ^{+} is join-uniform and $\mathbb{O}^{\downarrow}_{\eta,\rho}(f) \in \mathit{lco}(C \longrightarrow C)$ iff ρ is meet-uniform, where:

$$\mathbb{O}^{\uparrow}_{\eta,\rho}(f)(x) = \begin{cases} (\rho^{+})^{+}(f(x)) & \text{if } x \in \eta \\ f(x) & \text{otherwise} \end{cases}$$
$$\mathbb{O}^{\downarrow}_{\eta,\rho}(f)(x) = \begin{cases} \rho^{-}(f(x)) & \text{if } x \in \eta \\ f(x) & \text{otherwise} \end{cases}$$

All transformed semantics can be made monotone without afflicting minimality and completeness, see [29] for details. In this case, when they exist, $(\mathbb{O}^{\uparrow\downarrow}_{\eta,\rho})^{+} = \mathbb{F}^{\uparrow\downarrow}_{\eta,\rho}$.

7. Obscuring code

Software obfuscation provides protection against reverse-engineering, the goal of which is to understand programs. Lexical, data and control-flow obfuscation are typical examples of obfuscation strategies devoted to confuse the understanding of respectively lexical, data and control-flow of a given program.

An *obfuscating transformation*, according to Collberg *et al.*, is a program transformation $\mathfrak{O} : \mathbb{P} \longrightarrow \mathbb{P}$ such that 1) $\mathfrak{O}$ is a potent transformation, i.e., $\mathfrak{O}(P)$ is more *obscure* or *complex* that P and 2) P and $\mathfrak{O}(P)$ have the same observational behavior [7, 10, 11]. Potency is related with resiliency. $\mathfrak{O}$ is a resilient transformation if P is hardly obtainable from $\mathfrak{O}(P)$ by automatic transformation (i.e., by *deobfuscation*). While observational equivalence can be precisely encoded in programming language semantics, the notion of potency and resilience of an obfuscation are relatively unexplored notions, where qualitative and quantitative methods have been introduced, without a general agreement on how potency and resilience can be formally expressed.

Collberg *et al.*, [10] define potency as the relative complexity $\mathfrak{O}(P)$ with respect to P according to some known metrics such as code size, number of predicates, number of methods in OO programs, height of inheritance, and variable dependence length. Successful code obfuscation relying on these notions of potency include control-flow obfuscation by opaque predicate insertion, code flattening, variable splitting, bogus code insertion, and spurious aliases insertion. Wang *et al.* [50] relate potency with complexity of static program analysis, notably variable aliasing. The authors compile the problem of understanding control-flow into a general aliasing problem, which is NP-hard. Cloacked programs extend this approach by specifying code transformations (basically flattening + obfuscated dispatchers) such that the understanding of the obfuscated control-flow corresponds to the solution of a known (hard) combinatorial problem. Here potency is related with the PSPACE complexity of reachability in dispatchers [6]. In data-type obfuscation, potency is related with data-refinement [25]. An obfuscated program can be seen as a program with refined data-types, provided that a GI is established between source data-types and obfuscated ones. If D is a data-type, $\mathfrak{D}$ is a refinement of D if $\langle \mathfrak{D}, \alpha, \gamma, D \rangle$ is a GI. Correctness of the obfuscation is here proved by proving $[\![P]\!] = \alpha \circ [\![\mathfrak{O}(P)]\!] \circ \gamma$, i.e., by proving that the source program is the bca of the obfuscated one with respect to the data-type refinement abstraction. Here obfuscation corresponds precisely to concretise (in the sense of abstract interpretation) a data-type. Potency is not directly addressed in this framework, even if it can be related with the *distance* between the concrete and abstract data-types, as shown in [22]. This framework has been successfully applied for complicating code understanding by program slicing [37], where α and γ are programs which enlarge slices by adding dependencies and correctness is proved by requiring that P and $\gamma; \mathfrak{O}(P); \alpha$ are observationally equivalent, i.e., $[\![P]\!] = \alpha \circ [\![\mathfrak{O}(P)]\!] \circ \gamma$. Dalla Preda *et al.* [22, 21, 23] specify potency as the ability of $\mathfrak{O}(P)$ of *masking* some abstractions in the lattice of abstract interpretations. Here masking an abstraction means that there exists a program property $\rho \in \mathit{uco}(\Sigma^{*})$ such that $\rho([\![P]\!]) \neq \rho([\![\mathfrak{O}(P)]\!])$. The comparison, in the lattice of abstract interpretations, between masked abstractions and the most concrete abstract domain preserved by $\mathfrak{O}$ specifies its potency. Interestingly, in this framework, any program transformation can be interpreted as a program obfuscation with respect to some given property. Data and control-flow obfuscation by opaque predicate insertion have been specified in this framework. In particular, in [21], the authors proved that complete abstractions are essential to break opaque predicates in control-flow deobfuscation.

We follow [22], and refine that notion of potency with respect to the notion of interpretation. Understanding programs corresponds to understand their semantics. This is strictly connected with interpretation. Malicious host understand code by analysing (either dynamically or statically) the code. This corresponds precisely to perform an abstract interpretation, including the dynamic and static cases as instances with respectively non-decidable and decidable abstractions. In this case, $\mathfrak{O}(P)$ is an obfuscation of P if the abstract interpretation of $\mathfrak{O}(P)$ fails (is less precise) than the same abstract interpretation of P. Failing precision means failing completeness, therefore:

obfuscating programs is making abstract interpreters incomplete

The larger is the set of incomplete interpreters the stronger is the obfuscation. Let $\rho \in \mathit{uco}(\Sigma^*)$ be a property of the execution traces of programs in $\mathbb{P}$, and let $\mathfrak{O} : \mathbb{P} \longrightarrow \mathbb{P}$ be a program transformation, i.e., $[\![P]\!] = [\![\mathfrak{O}(P)]\!]$. Assume that ρ is $\mathcal{B}$-complete for $[\![P]\!]$, i.e., $\rho([\![P]\!]) = [\![P]\!]^\rho$. Then $\mathfrak{O}$ obfuscates P for ρ if

$$[\![P]\!]^\rho \sqsubset [\![\mathfrak{O}(P)]\!]^\rho$$

This is equivalent to say that $\rho([\![\mathfrak{O}(P)]\!]) \sqsubset [\![\mathfrak{O}(P)]\!]^\rho$, i.e., ρ is $\mathcal{B}$-incomplete for $[\![\mathfrak{O}(P)]\!]$. The following examples will clarify this idea and show how the basic abstract domain and semantics transformers for completeness can be used in this context.

Consider variable splitting used in slicing obfuscation [37]. Here obfuscation is performed by data-type refinement: variables $v \in \mathit{Var}(P)$ are split into pairs of variables $\langle v_1, v_2\rangle$, such that $v_1 = f_1(v)$, $v_2 = f_2(v)$ and $v = g(v_1, v_2)$. For instance, if $\mathbb{V} = \wp(\mathbb{Z})$:

$$\begin{gathered} f_1(v) = v \ \div\ 10 \\ f_2(v) = v \mod 10 \\ g(v_1, v_2) = 10 \cdot v_1 + v_2 \end{gathered}$$

Consider the following simple program P

```
P :  [  v = 0;
        while v < N  {v++}
```

and the corresponding obfuscated code $\mathfrak{O}(P)$

```
         [  v_1 = 0;
            v_2 = 0;
 O(P) :     while   10 · v_1 + v_2 < N {
                       v_1 = v_1 + (v_2 + 1) ÷ 10
                       v_2 = (v_2 + 1)   mod 10
                    };
     c :  v = 10 · v_1 + v_2
```

In this case, $\mathfrak{O}(P)$ obfuscates interval analysis. Consider the abstract domain $\iota \in \mathit{uco}(\wp([-\mathfrak{m}, \mathfrak{m}]))$ of limited intervals, where $\mathfrak{m} \in \mathbb{Z}$ is the maximal integer. In this case $\iota(x) = [\min(x), \max(x)]$. Interval analysis is defined in [14], with standard bca abstract interpretations for arithmetic operations on intervals: $\odot$, $\oplus$, $\ominus$. In this case the abstract collecting semantics for P is $[\![P]\!]^\iota = \lambda v.\ [0, N]$, while $[\![\mathfrak{O}(P)]\!]^\iota = \lambda\langle v_1, v_2\rangle.\ \langle[0, \frac{N-v_2}{10}], [0, 9]\rangle$ from which

$$\begin{aligned} [\![\mathfrak{O}(P); c]\!]^\iota &= \lambda v.\ 10 \odot [0, \tfrac{N \ominus [0,9]}{10}] \oplus [0, 9] \\ &= \lambda v.\ [0, N] \oplus [0, 9] \\ &= \lambda v.\ [0, N + 9] \end{aligned}$$

It is clear that the obfuscation here makes the interval abstract interpretation incomplete. This example shows that code obfuscation defeats abstract interpretation by weakening the generated invariants. Weak enough generated invariants make difficult the understanding of code, because strong invariants provide sufficient information to understand code behaviour. The role of weakened invariants by the malicious host analysis is essential in code obfuscation and models from this point of view the potency for the corresponding code transformation. *The weaker is the generated invariant the more obscure is the code*.

An element in the flat lattice of arrays is $a \in$ **array**$[\mathbb{Z}]$ where $a : D_a \longrightarrow \mathbb{Z}$ is an array with domain $D_a \subseteq \mathbb{N}$. Consider the following program computing the Fibonacci's sequence with $\mathbb{V} = \wp(\mathbb{Z}) \times$ **array**$[\mathbb{Z}]$ representing pairs of values for the integer variable i and integer arrays a, b, c.

```
      [  a[0] = 0;
         a[1] = 1;
         i = 2;
 P :     while   i ≤ N {
                   a[i] = a[i − 1] + a[i − 2];
                   i + +
                 }
```

and its obfuscation $\mathfrak{O}(P)$ by *array splitting* [25], which is a generalisation to arrays of variable splitting

```
$b[0] = 0;$
$c[0] = 1;$
$i = 2;$
while  $i \leq N$ {
    if  $i \mod 2 == 0$
        $\{b[i \div 2] = c[(i-1) \div 2] + b[(i-2) \div 2]\}$
        $\{c[i \div 2] = b[i \div 2] + c[(i-2) \div 2]\};$
         $i++$
        }
```

The potency of $\mathfrak{O}(P)$ is obtained by weakening the invariant **Inv** $= 2 \leq i \leq N \wedge \forall j \in [2, i].\ a[j] = a[j-1] + a[j-2]$ holding in the **while** statement of P. This invariant can be automatically generated by an abstract interpreter which performs relational static analysis on the abstract domain $\mathfrak{u} \xrightarrow{\bowtie} \eta$, where $\eta, \mathfrak{u} \in uco(\wp(\mathbb{V}))$, such that $\bowtie = \cap$, $\mathfrak{u}(X) = \left\{ \langle \iota(S), \top \rangle \mid \langle S, a \rangle \in X \right\}$ is the attribute independent extension of interval analysis to $\mathbb{V}$, and $\eta = \alpha^+ \circ \alpha$ where

$$\alpha(X) = \begin{cases} \textbf{Fib} & \text{if } \forall \langle S, x \rangle \in X.\ S \subseteq D_x \wedge \\ & \quad (S = \{0\} \wedge x[0] = 0) \vee \\ & \quad (S = \{0,1\} \wedge x[0] = 0 \wedge x[1] = 1) \vee \\ & \quad (\forall j \in S.\ x[j] = x[j-1] + x[j-2]) \\ \textbf{Any} & \text{otherwise} \end{cases}$$

isolates Fibonacci's sequences with in-bound indexes. Relevant objects in $\mathfrak{u} \xrightarrow{\bowtie} \eta$ can either be $I \longrightarrow \textbf{Fib}$, representing Fibonacci's sequences until $\max(I)$ or $I \longrightarrow \textbf{Any}$, representing any array with domain including I (no overflow). The abstract interpretation of P in $\mathfrak{u} \xrightarrow{\bowtie} \eta$ is based on the bca of the basic composition $\oplus$ for Fibonacci's sequences:

$$[n, m] \longrightarrow \textbf{Fib} \overset{\text{def}}{=} [n, m-1] \longrightarrow \textbf{Fib} \oplus [n, m-2] \longrightarrow \textbf{Fib}$$

In this case we have:

$$\begin{aligned} \llbracket P \rrbracket^{\mathfrak{u} \xrightarrow{\bowtie} \eta} &= a \in [0, N] \longrightarrow \textbf{Fib} \wedge i \in [2, N+1] \\ \llbracket \mathfrak{O}(P) \rrbracket^{\mathfrak{u} \xrightarrow{\bowtie} \eta} &= b, c \in [0, N \div 2] \longrightarrow \textbf{Any} \wedge i \in [2, N+1] \end{aligned}$$

The obfuscation here breaks the coherence of the invariant property detected by η, by splitting the original array into two arrays which are not Fibonacci's sequences. The original invariant **Inv** can only be reconstructed by refining the analysis with relational information between the new arrays (b and c). The complete shell $\mathcal{R}^{\mathcal{B}}_{\llbracket \mathfrak{O}(P) \rrbracket}(\mathfrak{u} \xrightarrow{\bowtie} \eta)$ is indeed able to isolate arrays having odd-(even-)positioned Fibonacci's numbers. Understanding this refinements corresponds precisely to deobfuscate $\mathfrak{O}(P)$.

In view of this approach to code obfuscation and of the transformations developed in [29], it is possible to maximally obfuscate a given semantics by inducing maximal incompleteness in code instructions. The idea is to defeat a given abstraction by transforming code instructions in order to make them maximally incomplete for that abstration. Let us introduce this idea by an example. It is easy to prove that the limited-interval abstract domain is a meet-uniform closure: if $Y \subseteq \wp([-\mathfrak{m}, \mathfrak{m}])$ and for any $x, y \in Y$: $\iota(x) = \iota(y)$, then for any $x, y \in Y$. $\min(x) = \min(y) \wedge \max(x) = \max(y)$. Therefore there exists $z \in Y$ such that $\iota(\bigcap Y) = \iota(z)$. In this case, $\iota^- = \lambda x.\ \{\min(x), \max(x)\} \in lco(\wp([-\mathfrak{m}, \mathfrak{m}]))$. This observation may drive us towards the systematic design of obfuscated code against interval analysis. Consider the following simple program.

```
P : [  c :  $x = x * x;$
            if $10 \leq x \leq 100$ {$y = 5$} {$y = 5000$};
            return($y$)
```

The forward limited interval analysis of P with $x = [5, 8]$ returns: $\llbracket P \rrbracket^{\iota}(x \in [5, 8]) = x \in [25, 64] \wedge y \in [5]$. Note that $\textbf{wlp}\llbracket c \rrbracket^{\iota}(y \leq 100) = x \in [10, 100]$ moreover we have $\textbf{wlp}\llbracket x = x * x \rrbracket^{\iota}(x \in [10, 100]) = x \in [4, 10]$. Therefore, in order to make interval analysis $\mathcal{B}$-incomplete before program point c, we can define a command c' such that

$$\begin{aligned} &\textbf{wlp}\llbracket c' \rrbracket^{\iota}(x \in [10, 100]) = \\ &\qquad \mathbb{O}^{\downarrow}_{\iota, \iota}(\lambda X.\ \textbf{wlp}\llbracket x = x * x \rrbracket^{\iota}(X))(x \in [10, 100]) \end{aligned}$$

In this case, $\iota^-(\textbf{wlp}\llbracket x = x*x \rrbracket^{\iota}(x \in [10, 100])) = \{4, 10\}$. The following command:

$$c' : \textbf{if}\ x == 4 \vee x == 10\ \{x = 16\}\ \{x = x * 200\}$$

satisfies this condition. Clearly we have to cope with intermediate values in such a way that the variable y is set to 5 for all $x \in [4, 10]$, keeping in this way the behavioural equivalence. This is achieved by implementing the ι^- closure. Also this implementation has to exploit incompleteness in such a way that its output interval may activate both branches of c'. The following command, obfuscated with simple equivalent instruction sequences, implements ι^- with these features for all relevant inputs $x \in [4, 10]$:

```
if  $4 \leq x \leq 10$
    $\{x = x - (x - 4) \ \square\ x = x - (x - 10)\}$
    {nil}
```

The resulting obfuscated code is:

$$\mathfrak{O}(P) : \left[\begin{array}{l} \textbf{if} \quad 4 \leq x \leq 10 \\ \quad \{x = x - (x-4) \,\Box\, x = x - (x-10)\} \\ \quad \{\textbf{nil}\}; \\ \textbf{if}\ x == 4 \vee x == 10\ \{x = 16\}\ \{x = x * 200\}; \\ \textbf{if}\ 10 \leq x \leq 100\ \{y = 5\}\ \{y = 5000\}; \\ \textbf{return}(y) \end{array}\right.$$

In this case the maximal incomplete transformation of c requires the analogous transformation of all instructions from which c depends, i.e., the backward slice of c in P. The limited interval analysis of $\mathfrak{O}(P)$ starting with $x = 7$ is: $[\![\mathfrak{O}(P)]\!]^{\iota}(x \in [5, 8]) = x \in [16, 1400] \wedge y \in [5, 5000]$. The key point here is the ability of command c' to make incomplete the evaluation of the test in c, causing the loss of precision in the analysis of variable y. This obfuscation schema can be used for generating obfuscated code by systematic transformations of semantics, in particular when dealing with opaque predicates and control-flow obfuscation. In this case, the maximal incomplete transformers may help providing obscure opaque predicates. Breaking them requires more complex refined abstract domains and analysis. In the case of the example above, the refined analysis needs either to discover constant expressions (e.g., $x - (x - 4)$ and $x - (x - 4)$) or to perform some form of disjunctive completion in interval analysis, the latter being extremely expensive ($\curlyvee(\iota) = \mathit{id}$).

8. Hiding information

Among the different methods for hiding secrets in programs, software watermarking is one of the most common. We consider a steganographic approach to software watermarking, i.e., program transformations where the intended (typically copyright) signature is hidden from external observers. We follow [19] by defining the *stegomarker* $\mathfrak{M} : \mathbb{S} \longrightarrow \mathbb{P}$ as the encoding of the signature $s \in \mathbb{S}$ into a program $\mathfrak{M}(s) \in \mathbb{P}$, called the *stegomark*. The *stegolayer* $\mathfrak{L} : \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{P}$ is used to compose the stegomark with the source (cover) program. The *(watermarked) stegoprogram* is $\mathfrak{S} : \mathbb{P} \times \mathbb{S} \longrightarrow \mathbb{P}$ such that $\mathfrak{S}(P, s) = \mathfrak{L}(P, \mathfrak{M}(s))$. The standard taxonomy of software watermarking in [8, 9, 41] distinguish between *static watermarking*, where signatures are encoded as properties of the code text, and *dynamic watremarking*, where the signature is encoded in the state computed by the stegoprogram under suitable inputs. Abstract watermarking, introduced in [19], is different: the signature is encoded as a stegomark in the cover program and can be extracted by suitable static program analysis.

We believe that static and dynamic watermarking are instances of abstract watermarking, under suitable choices for $\mathfrak{M}$ and $\mathfrak{L}$. In particular, they are instances of a common pattern which corresponds precisely to the program transformations making semantics complete [29]. Let $P \in \mathbb{P}$ and $\alpha, \omega, \eta \in \mathit{uco}(\Sigma^*)$ be program properties such that $\alpha \sqsubseteq \omega$. If $(\!|\mathfrak{M}(s)|\!)^{\alpha} \in \omega$ then $\mathfrak{L}$ is a stegolayer for P and $\mathfrak{M}(s)$ if

$$(\!|\mathfrak{L}(P, \mathfrak{M}(s))|\!)^{\alpha} \stackrel{\text{def}}{=} \lambda x. \begin{cases} (\!|\mathfrak{M}(s)|\!)^{\alpha}(x) & \text{if } x \in \eta \\ (\!|P|\!)^{\alpha}(x) & \text{otherwise} \end{cases}$$

Static software watermarking corresponds to $\eta = \mathit{id}$, i.e., $\forall x.\ (\!|\mathfrak{S}(s, P)|\!)^{\alpha}(x) = (\!|\mathfrak{M}(s)|\!)^{\alpha}(x) \in \omega$, and α decidable. This means that the interpretation of the stegoprogram always reveals the watermark, independently from the input. In dynamic watermarking instead $\eta \neq \mathit{id}$, meaning that only suitable inputs may reveal the watermark. In this case, the inputs revealing the watermark are all the inputs satisfying η. In this context, the syntactic stegomarker $\mathfrak{M}(\cdot)$ can be associated with a semantic stegomarker $\mathfrak{M}(\!|\cdot|\!) : \mathbb{S} \longrightarrow \mathit{uco}(\Sigma^*)$. It is immediate to recognise a $\mathcal{F}$-completeness transformation here: *A stegoprogram reveals the watermark ω under input η if its abstract semantics is $\mathcal{F}$-complete for ω and η.* The abstract semantics $(\!|\cdot|\!)^{\alpha}$ performs watermark extraction which is, as in [19], implemented as abstract interpretation. Therefore $\mathfrak{S}(s, P)$ is a stegoprogram if:

$$(\!|\mathfrak{S}(s, P)|\!)^{\alpha} = \mathbb{F}^{\uparrow\downarrow}_{\eta, \mathfrak{M}(\!|s|\!)}((\!|P|\!)^{\alpha})$$

Note that if $\langle \omega, \eta \rangle \in \mathcal{F}(\wp(\Sigma^*), (\!|\mathfrak{S}(s, P)|\!)^{\alpha})$ it may well happen that $\langle \omega, \eta \rangle \notin \mathcal{F}(\wp(\Sigma^*), (\!|\mathfrak{S}(s, P)|\!))$ [34, 29]. This means that the knowledge of the stegomarker may not be sufficient in order to extract the watermark. This makes the extraction completely dependent on the suitable choice of the abstract semantics $(\!|\cdot|\!)^{\alpha}$. In this sense, code obfuscation can be used in order to design appropriate stegolayers making $\langle \omega, \eta \rangle$ incomplete for the standard interpreter $(\!|\cdot|\!)$. This is a further weakening with respect to abstract watermarking [19], where $\omega = \alpha$ and the secrecy relies upon the difficulty to guess ω out of any blind static or dynamic analysis of the stegoprogram.

Credibility, *data-rate*, and *resilience* [9] rely upon the choice of the properties α and ω. High credibility corresponds to $\alpha, \omega \in \mathit{uco}(\Sigma^*)$ such that $(\!|P|\!)^{\alpha} \notin \omega$ (i.e., $\omega((\!|P|\!)^{\alpha}) \approx \top$ minimises false positives). Resilience is high when ω, and therefore α, are both hard to guess and they are preserved by most common program transformations. *Stealthy* instead depends upon the implementation of the stegolayer, which has to produce output code which is as similar as possible to P. Note that, being $\mathbb{F}^{\downarrow}$ and

$\mathbb{F}^{\uparrow}$ idempotent transformations, they provide also a code tamper-detection method similar to the one used for images in mathematical morphology [35]. In this case, because $\mathbb{F}^{\uparrow\downarrow}_{\eta,\mathfrak{M}(\!|s|\!)}((\!|\mathfrak{S}(s,P)|\!)^{\alpha}) = (\!|\mathfrak{S}(s,P)|\!)^{\alpha}$, then any malicious host attack (e.g., distortive) transformation $t : \mathbb{P} \longrightarrow \mathbb{P}$ such that $\mathbb{F}^{\uparrow\downarrow}_{\eta,\mathfrak{M}(\!|s|\!)}((\!|t(\mathfrak{S}(s,P))|\!)^{\alpha}) \neq (\!|t(\mathfrak{S}(s,P))|\!)^{\alpha}$ will reveal the attack.

It is easy to encode within this schema most well-known watermarking methods. We sketch some of these encoding for popular watermarking methods. *Abstract watermarking* [18] is immediate. *Block reordering:* this static watermarking corresponds to the following choices: $\eta = id$ (static); given a signature (number) s and an encoding of numbers in graphs as sequences of basic blocks $\mathcal{E} : \mathbb{N} \longrightarrow \mathbb{G}$ then $\mathfrak{M}(\!|s|\!)$ is the atomic closure $\{\mathcal{G}_s, \Sigma^*\} \in uco(\Sigma^*)$ where $\mathcal{G}_s = \left\{ \sigma \in \Sigma^* \,\middle|\, \mathcal{E}(s) = \mathrm{CFG}(\sigma) \right\}$ and $(\!|P|\!)^{\alpha}$ extracts the CFG of P, which is an (incomplete) abstract interpretation of the trace semantics $(\!|P|\!)$ provided that states include code instructions with labels. The abstraction α forgets about memory locations and computed values and just keeps track of the sequence of program instructions isolating basic blocks (consecutive instructions) as graph nodes and determining possible jumps between blocks as graph edges [47]. A dynamic version of block reordering can be implemented by choosing $\eta \neq id$. In this case $\mathfrak{S}(s,P)$ has to include a block reordering algorithm, which is activated when $x \in \eta$, as in metamorphic malware [20]. The same abstractions can be used for encoding Venkatesan *et al.* CFG-based watermarking [49]. *Constraint-based watermarking:* we consider the graph-coloring static (non-blind) watermarking solution in [44], applied to register allocation. In this case program states include register allocation mappings $\mathfrak{R} : Var(P) \longrightarrow \mathscr{R}$, where $\mathscr{R}$ is a given (fixed) finite set of registers. Elements in Σ are $\langle c, \mathfrak{R}, \mathcal{H}\rangle$, where $\mathcal{H} \in \mathbb{H}$ is a heap and c is the current instruction. $\mathfrak{R}$ is statically computed by a pre-processing phase. In this case $\eta = id$, and $\alpha = \beta^{+} \circ \beta$ where $\beta : \wp(\Sigma^*) \xrightarrow{\mathrm{m}} \wp(Var(P) \times Var(P))$:

$$\beta(X) \stackrel{\text{def}}{=} \left\{ \langle v_1, v_2\rangle \;\middle|\; \begin{array}{l} \mathfrak{R}(v_1) = \mathfrak{R}(v_2) \\ \langle c_0, \mathfrak{R}, \mathcal{H}_0\rangle \rightsquigarrow^{*} \langle c_n, \mathfrak{R}, \mathcal{H}_n\rangle \in X \end{array} \right\}$$

α is decidable and it extracts the interference graph associated with a given register allocation, as an abstract interpretation of the trace semantics. *Graph-based watermarking:* the dynamic graph-based watermarking encodes the watermark in a suitable data-structure which is allocated in memory [12, 42]. In this case, programs states are as above, including the sequence of input values $i \in \mathbb{V}^*$ which still have to be consumed: $\langle c, \mathfrak{R}, \mathcal{H}, i\rangle$, $\eta = \{\mathfrak{I}, \Sigma^*\}$ where $i \in \mathbb{V}^*$ is a given input sequence, $\omega = \{\mathcal{E}(s), \Sigma^*\}$ where $\mathcal{E}(s)$ is the encoding of s as a graph $\mathcal{G}_s \in \mathbb{G}$. α observes the graphs encoded in memory, by looking at graphs in reverse allocation order. In this case $\alpha = \delta^{+} \circ \delta$ where $\mathfrak{H} : \mathbb{H} \longrightarrow \mathbb{G}$ extracts the set of all graphs allocated in memory with **root** allocated as last, and $\delta : \wp(\Sigma^*) \xrightarrow{\mathrm{m}} \mathbb{G}$ is such that:

$$\delta(X) \stackrel{\text{def}}{=} \left\{ \mathcal{G} \;\middle|\; \begin{array}{l} \sigma \in X,\ |\sigma| = n+1,\ \sigma_n = \langle c, \mathfrak{R}, \mathcal{H}_n, \varepsilon\rangle \\ \mathcal{G} \in \mathfrak{H}(\mathcal{H}_n),\ \mathbf{root}(\mathcal{G}) \in \mathcal{H}_n \\ \forall j \in [0, n-1].\ \mathbf{root}(\mathcal{G}) \notin \mathcal{H}_j \end{array} \right\}$$

Threading watermarking [40] would need a different computational model, including multithreading and concurrency. In this case the extractors should correspond to a complete abstract interpretation modelling execution paths, which encode the watermark.

9. Finding completeness holes

Obscuring code and hiding information are different aspects of the same issue, which is making an interpreter incomplete, either by transforming the source code by obfuscation or by designing suitable code (the stegoprogram) whose interpretation is incomplete unless some key properties (the secret watermark extractor) are known. In both cases, a basic program transformation is defined: $\mathfrak{O}$ for obfuscation and $\mathfrak{M}$ for specifying the stegomark. The transformed program is then integrated with other (non affected) code by a suitable program integration method, which is the stegolayer $\mathfrak{L}$ in software watermarking and it is standard sequential composition in most code obfuscation. We generalise this situation by considering a generic (binary associative) program integration method $\mathfrak{I} : \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{P}$ [46]. The obfuscation of code parts and the stegomarks have to preserve the observable semantics of the original program when the transformed code is integrated by $\mathfrak{I}$ with a cover program. As observed above, e.g., in the last example of Section 7, the code transformation may affect the semantics of internal variables (x) provided that the observable behaviour (y) is not affected by this transformation. This can be precisely captured by *abstract non-interfernce* [27]. Abstract non-interference (ANI) [27] is a natural weakening of non-interference by abstract interpretation. Consider a partition of values/states $\Sigma = \mathbb{V} = \mathbb{V}^{\mathtt{L}} \times \mathbb{V}^{\mathtt{H}}$ in public $\mathtt{L}$ and private $\mathtt{H}$ values. We consider a pair of external observations $\eta, \rho \in uco(\mathbb{V}^{\mathtt{L}})$ (the attacker) and $\phi \in uco(\mathbb{V}^{\mathtt{H}})$ (the secret) which specifies which property of the private data cannot flow to the output observation. Recall that a program P satisfies (blocked) ANI, $(\eta)P(\phi \rightsquigarrow [\![\rho)$, if

$\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}}$ [27]:

$$\eta(l_1) = \eta(l_2) \implies \\ \rho(\llbracket P \rrbracket^{\phi,\eta}(\phi(h_1), \eta(l_1))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket^{\phi,\eta}(\phi(h_2), \eta(l_2))^{\mathtt{L}})$$

This notion says that, whenever the attacker is able to observe the input property η and the ρ property of the output, then no information flow concerning the property ϕ of the secret input interferes with the output observation ρ. In order to model secrecy in code transformations we consider a higher-order version of ANI, called HOANI, which shares with ANI all relevant properties [27, 28]. Here programs in $\mathbb{P}$ are partitioned in *cover programs* $\mathcal{P} \subseteq \mathbb{P}$ and secret programs $\mathcal{Q} \subseteq \mathbb{P}$. The cover (unaffected) program plays the role of the public input, the private input is the secret code whose properties have to be kept secret by the program integration method $\mathfrak{I}$. The pair $\langle \eta, \rho \rangle$ specifies here the cover/output observable, which is in our case the attacker. Let $\rho, \eta, \phi \in \mathit{uco}(\wp(\Sigma))$ and $\rho^{\rho} \in \mathit{uco}(\wp(\Sigma) \xrightarrow{\text{m}} \wp(\Sigma))$ such that $\rho^{\rho} \stackrel{\text{def}}{=} \lambda f.\ \rho \circ f \circ \rho$ [17]. Let $P_1, P_2 \in \mathcal{P}$ be (public) cover programs and $Q_1, Q_2 \in \mathcal{Q}$ be (secret) programs, e.g., holding a watermark or breaking an invariant at some given program point for obfuscation. Then $(\eta)\mathfrak{I}(\phi \rightsquigarrow\!\| \rho)$ holds in (denotational-based) HOANI if for any of these programs:

$$\llbracket P_1 \rrbracket^{\eta} = \llbracket P_2 \rrbracket^{\eta} \implies \\ \rho^{\rho}(\llbracket \mathfrak{I} \rrbracket^{\phi,\eta}(\llbracket Q_1 \rrbracket^{\phi}, \llbracket P_1 \rrbracket^{\eta})) = \rho^{\rho}(\llbracket \mathfrak{I} \rrbracket^{\phi,\eta}(\llbracket Q_2 \rrbracket^{\phi}, \llbracket P_2 \rrbracket^{\eta}))$$

In code obfuscation, $\mathcal{Q} = \{\mathfrak{O}(P), P\}$, $\mathcal{P} \subseteq \mathbb{P}$ and we want that $(\rho)\mathfrak{I}(\phi \rightsquigarrow\!\| \rho)$ holds for any ϕ such that $\llbracket \mathfrak{O}(P) \rrbracket^{\phi} \neq \llbracket P \rrbracket^{\phi}$. In software watermarking instead, $\mathcal{Q} = \left\{\, \mathfrak{M}(s) \,\middle|\, s \in \mathcal{S} \,\right\}$, $\mathcal{P} \subseteq \mathbb{P}$, and we want that $(\rho)\mathfrak{I}(\omega \rightsquigarrow\!\| \rho)$ holds for the largest possible set of ρ such that $\rho \neq \alpha$. In this case the systematic derivation of the strongest harmless attackers $\bar{\rho}$ such that $(\bar{\rho})\mathfrak{I}(\omega \rightsquigarrow\!\| \bar{\rho})$ [27] may provide a useful measure of the secrecy of the stegomark.

10. Discussion

In this paper we studied completeness in the context of code obfuscation and watermarking. This is different with respect to the notion of *obfuscated interpretations* in [39]. In this latter case the obfuscating transformation is applied to the interpreter in such a way that the interpretation of a command c is not fixed. This provides amazing results in terms of software protection via obscurity. We believe that also this idea can be fully specified and studied in terms of completeness of an abstract interpretation. The influence of data-type refinement approach to code obfuscation [25] in our approach is clear. While completeness methods provide driving guidelines for specifying and evaluating obfuscation and watermarking tools, the data-type refinement, which can be clearly specified in abstract interpretation form, provides the basis for proving correctness of the transformations. We believe that both methods should be included into any comprehensive theory of information hiding in code. From this point of view, we believe that HOANI is more general than data-type refinement based correctness methods [25], providing both correctness and secrecy in a unique setting. From a more practical point of view, the use of logic for information flow [2] may provide useful interference information between the variables involved in obfuscated/watermarked code and the cover program. As future work we are interested in extending the calculational design of program transformations by abstract interpretation in [18] to the calculational design of obfuscated code, as in [22], including completeness transformers in the framework. We are also interested in deriving appropriate metrics for estimating the quality of an obfuscation and watermarking method. The completeness-based approach to information hiding may provide here useful metrics, such as the one in [1, 24] measuring the degree of information leakage, which is known to be strongly related with the incompleteness of an abstract interpretation [28]. Of particular interest could be exploiting incompleteness holes derived from the inaccurate propagation of roundoff errors in floating-point operations [38]. This can be exploited, provided that inaccuracy is confined in the sense of ANI, in particular in advanced numerical software watermarking.

Acknowledgements. This paper has been conceived during my visit to the École Normale Supérieure, in Paris, in the summer of 2008. I would like to thank Patrick and Radhia Cousot for the kind hospitality. I would like also to thank Mila Dalla Preda, Isabella Mastroeni, and Enrico Visentini for the valuable discussions we had in the fall of 2007, discussions from which these ideas originated.

References

[1] A. Aldini and A. Di Pierro. Estimating the maximum information leakage. *Int. J. Inf. Sec.*, 7(3):219–242, 2008.

[2] T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*, 64(1):3–28, 2007.

[3] R. Andesron and F. Petitcolas. On the limits of steganography. *IEEE J. of Selected Areas in Communications*, 16(4):474–481, 1998.

[4] 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, London, UK, 2001. Springer-Verlag.

[5] T. Blyth and M. Janowitz. *Residuation theory*. Pergamon Press, 1972.

[6] 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, London, UK, 2001. Springer-Verlag.

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

[8] 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, New York, NY, USA, 1999. ACM.

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

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

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

[12] C. Collberg, C. D. Thomborson, and G. M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6):35, 2007.

[13] 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.

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

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

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

[17] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages) (Invited Paper). In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '94*), pages 95–112, Los Alamitos, Calif., 1994. IEEE Comp. Soc. Press.

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

[19] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, 2004. ACM Press, New York, NY.

[20] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2007. ACM Press.

[21] 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, Washington, DC, USA, 2005. IEEE Computer Society.

[22] M. Dalla Preda and R. Giacobazzi. Semantics-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.

[23] M. Dalla Preda, M. Madou, K. D. 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, Berlin, 2006. Springer-Verlag.

[24] A. Di Pierro, C. Hankin, and H. Wiklicky. Measuring the confinement of probabilistic systems. *Theor. Comput. Sci.*, 340(1):3–56, 2005.

[25] S. Drape. *Obfuscation of Abstract Data-Types*. PhD thesis, University of Oxford, 2004.

[26] G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Comput. Surv.*, 28(2):333–336, 1996.

[27] 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, New York, 2004. ACM-Press.

[28] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In S. Sagiv, editor, *Proc. of the European Symp. on Programming (ESOP '05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310, Berlin, 2005. Springer-Verlag.

[29] 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.

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

[31] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. of the 24th Internat. Colloq. on Automata, Languages and Programming* (*ICALP '97*), volume 1256 of *Lecture Notes in Computer Science*, pages 771–781, Berlin, 1997. Springer-Verlag.

[32] 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.

[33] R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theor. Comput. Sci*, 216:159–211, 1999.

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

[35] M. Kihara, M. Fujiyoshi, Q. T. Wan, and H. Kiya. Image tamper detection using mathematical morphology. In *ICIP 2007: IEEE International Conference on Image Processing*, pages 101–104. IEEE, 2007.

[36] 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, New York, NY, USA, 2003. ACM.

[37] 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, New York, NY, USA, 2007. ACM.

[38] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher Order Symbol. Comput.*, 19(1):7–30, 2006.

[39] A. Monden, A. Monsifrot, and C. D. Thomborson. A framework for obfuscated interpretation. In *ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 7–16, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[40] J. Nagra and C. D. Thomborson. Threading software watermarks. In *In the proceedings of 6 th International Workshop on Information Hiding*, volume 3200 of *Lecture Notes in Computer Science*, pages 208–233. Springer-Verlag, 2004.

[41] J. Nagra, C. D. Thomborson, and C. Collberg. A functional taxonomy for software watermarking. *Aust. Comput. Sci. Commun.*, 24(1):177–186, 2002.

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

[43] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding – A survey. *Proc. of the IEEE*, 87(7):1062–1078, 1999.

[44] G. Qu and M. Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 190–193. ACM Press, 1998.

[45] F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proc. of the 13th European Symp. on Programming (ESOP '04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32, Berlin, 2004. Springer-Verlag.

[46] T. Reps. Algebraic properties of program integration. *Sci. Comput. Program.*, 17:139–215, 1991.

[47] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007.

[48] K. I. Rosenthal. Quantales and their applications. In *Pitman Research Notes in Mathematics*. Longman Scientific & Technical, London, 1990.

[49] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Information Hiding*, volume 2137 of *Lecture Notes in Computer Science*, pages 157–168, 2001.

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

[51] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, Cambridge, Mass., 1993.