

A semantics-based approach to Software Watermarking by Abstract Interpretation

Mila Dalla Preda and Michele Pasqua
(mila.dallapreda|michele.pasqua)@univr.it

Technical Report number: RR 98/2016
Università degli studi di Verona - Dipartimento di Informatica
This report is available at: <http://www.di.univr.it/report>

17/02/2016

Abstract

Software watermarking is a defence technique used to prevent software piracy by embedding a signature in the code. When an illegal copy is made, the ownership can be claimed by extracting the signature. The signature has to be hidden inside the code and it has to be difficult for an attacker to detect, tamper or remove it. In this paper we show how the ability of the attacker to identify the signature can be modelled in the framework of abstract interpretation as a completeness property. We view attackers as abstract interpreters that can precisely observe only the properties for which they are complete. In this setting, hiding a signature in the code corresponds to insert it in terms of a semantic property that can be retrieved only by attackers that are complete for it. Indeed, any abstract interpreter that is not complete for the property specifying the signature cannot detect, tamper or remove it.

Contents

1	Introduction	4
2	Preliminaries	5
2.1	Basic notions and mathematical notation	5
2.2	Abstract interpretation	6
2.2.1	Inducing forward completeness	7
2.3	Programming language and semantics	7
2.3.1	Semantics of programs	11
2.3.2	Abstract semantics	15
2.4	Abstract non-interference	16
2.4.1	Deriving attackers	17
3	Formalization	19
3.1	The framework	21
3.2	Properties	23
3.2.1	Resilience	24
3.2.2	Secrecy	27
3.2.3	Transparency	28
3.2.4	Accuracy	29
4	Validation	31
4.1	Static graph-based watermarking	31
4.2	Path-based watermarking	33
4.3	Abstract constant propagation watermarking	36
4.4	Block-reordering watermarking	39
4.5	Dynamic graph-based watermarking	41
5	Comparison	44
5.1	Resilience	44
5.1.1	Edge-flipping	44
5.1.2	Opaque predicate insertion	45
5.1.3	Dead code elimination	46
5.1.4	Loop-unrolling	47
5.1.5	Loop-invariant code motion	48
5.2	Secrecy and Transparency	50
6	Conclusions	51
	References	52
	Appendices	55
A	Algorithms	55

1 Introduction

Software developers are interested in protecting the intellectual property of their products against software piracy, namely to prevent the illegal reuse of their code. Code obfuscation, whose aim is to obstruct code decipherment, represents a preventive tool against software piracy: attackers cannot steal what they do not understand [1]. Once an attacker goes beyond this defense, software watermarking allows the owner of the violated code to prove the ownership of the pirated copies [D96, 3, 2, 20]. Software watermarking is a technique for embedding a signature, i.e., an identifier reliably representing the owner, in a cover program. This allows software developers to prove their ownership by extracting their signature from the pirated copies. In recent years researchers have developed a variety of software watermarking techniques (e.g., [3, 2, 21]) that can be classified in three main categories according to their extraction process: static, dynamic and abstract watermarking. *Static watermarking* inserts signatures in the cover program either as data or code and then extracts them statically, namely without executing the code [2]. Conversely, *dynamic watermarking* inserts signatures in the program execution state (i.e., in its semantics) and the extraction process requires the execution of the program, often on a special enabling input [2]. *Abstract watermarking*, introduced in [8], encodes the signature in such a way that it could be extracted only by a suitable abstract execution of the program. A watermarking scheme is typically evaluated w.r.t. the following features: credibility (how strongly it proves authorship), secrecy (how difficult it is to extract the mark), transparency (how difficult it is to realize that a program is marked), accuracy (observational equivalence of the marked and original program), resilience to attacks (how difficult it is to compromise the correct extraction of the signature) and data-rate (amount of information that can be encoded). The quality of each existing watermarking technique is specified in terms of these features that are typically claimed to hold w.r.t. the peculiar embedding and extraction methods. There exists a variety of embedding and extraction algorithms that often work on different object (control flow graph, variables, registers, etc.) and this makes it difficult to compare the efficiency of different watermarking systems. It is therefore difficult to formally prove and compare limits and potentialities of the different watermarking systems and to decide which one is better to use in a given scenario.

A first attempt to provide a formal definition of a watermarking system has been proposed in [15]. Here the author introduced the idea that static, dynamic and abstract watermarking could be seen as particular instances of a common watermarking scheme based on program semantics and abstract interpretation. In this work we start from the formal watermarking framework proposed in [15] and we extend and validate it. The idea is to model the embedding of the secret signature s as the encoding of s as a

semantic property $\overline{\mathfrak{M}}(s)$ that is then inserted in the semantics of the cover program. In this setting, the extraction process requires an analysis of the marked code that has to be at least as precise as $\overline{\mathfrak{M}}(s)$. This notion of precision of the extraction corresponds to the notion of completeness of the analysis in abstract interpretation. This means that in order to extract the signature it is necessary to know how it is encoded. In this view the semantic property for which the analysis has to be complete in order to extract the signature plays the role of an extraction key. Indeed, the signature results hidden for any observer of the program semantics that it is not complete for $\overline{\mathfrak{M}}(s)$, namely that does not know the secret key. Based on these ideas we provide a formal semantics-based definition of a watermarking system. Moreover, we provide a specification of the features of a watermarking system in the semantic framework in terms of semantic program properties. For example, it turns out that a watermarking scheme is transparent w.r.t. and observer when the embedding process preserves the program properties in which the observer is interested. Moreover, the resilience of a watermarking scheme to collusive attacks, that attempt to remove the signature by comparing different marked programs, can be modeled as a property of abstract non-interference among programs.

Our investigation and study in this direction has led to the following contributions.

- Specification of a formal framework based on program semantics and abstract interpretation for the modeling of software watermarking. The framework stabilizes the one proposed in [15].
- Formalization of the quality features (resilience, secrecy, transparency, accuracy) used to measure the quality of a watermarking system in the framework.
- Validation of the framework on five watermarking techniques together with a qualitative comparison of their quality features.

2 Preliminaries

2.1 Basic notions and mathematical notation

Given two sets S and T , we denote with $\wp(S)$ the powerset of S , with $S \setminus T$ the set-difference between S and T , with $S \subset T$ strict inclusion and with $S \subseteq T$ inclusion. Let S_{\perp} be set S augmented with the *undefined value* \perp , i.e., $S_{\perp} = S \cup \{\perp\}$. $\langle P, \leq \rangle$ denotes a poset P with ordering relation \leq , while a complete lattice P , with ordering \leq , least upper bound (lub) \vee , greatest lower bound (glb) \wedge , greatest element (top) \top , and least element (bottom) \perp is denoted by $\langle P, \leq, \vee, \wedge, \top, \perp \rangle$. Often, \leq_P will be used to denote the underlying ordering of a poset P , and \vee_P, \wedge_P, \top_P and \perp_P denote the basic

operations and elements of a complete lattice P . \sqsubseteq denotes pointwise ordering between functions. If $f : S \rightarrow T$ and $g : T \rightarrow Q$ then $g \circ f : S \rightarrow Q$ denotes the composition of f and g , i.e., $g \circ f = \lambda x.g(f(x))$. $f : P \rightarrow Q$ on posets is (Scott)-continuous when f preserves lub of countable chains in P . $f : C \rightarrow D$ on complete lattices is additive (co-additive) when for any $Y \subseteq C$, $f(\vee_C Y) = \vee_D f(Y)$ ($f(\wedge_C Y) = \wedge_D f(Y)$).

2.2 Abstract interpretation

Abstract interpretation is based on the idea that the behaviour of a program at different levels of abstraction is an approximation of its (concrete) semantics [6, 7]. The concrete program semantics is computed on the concrete domain $\langle C, \leq_C \rangle$, while approximation is encoded by an abstract domain $\langle A, \leq_A \rangle$. In abstract interpretation abstraction is specified as a Galois connection (GC) (C, α, γ, A) , namely as an abstraction map $\alpha : C \rightarrow A$ and a concretization map $\gamma : A \rightarrow C$ that are monotone and that form an adjunction: $\forall y \in A, x \in C : \alpha(x) \leq_A a \Leftrightarrow x \leq_C \gamma(y)$ [6, 7]. α [resp. γ] is the left- [right] adjoint of γ [α] and it is additive [co-additive], i.e. it preserves the lub [glb] of all the subsets of the domain (emptyset included). Abstract domains can be equivalently formalized as upper closure operators on the concrete domain [7]. The two approaches are equivalent, modulo isomorphic representations of the domain object. An upper closure operator, or closure, on poset $\langle C, \leq \rangle$ is an operator $\varphi : C \rightarrow C$ that is monotone, idempotent and extensive (i.e. $\forall c \in C : c \leq \varphi(c)$). Closures are uniquely determined by the set of their fixpoints $\varphi(C)$. The set of all upper closure operators on C is denoted by $uco(C)$. The lattice of abstract domains of C , is therefore isomorphic to $uco(C)$ [6, 7]. Recall that if C is a complete lattice, then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, id \rangle$ is a complete lattice, where $id \stackrel{\text{def}}{=} \lambda x.x$ and for every $\rho, \eta \in uco(C)$, $\rho \sqsubseteq \eta$ iff $\forall y \in C : \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$. Given $X \subseteq C$, the least abstract domain containing X is the least closure including X as fix-points, which is the Moore-closure $\mathcal{M}(X) \stackrel{\text{def}}{=} \{\vee S \mid S \subseteq X\}$. If (C, α, γ, A) is a GC then $\varphi = \gamma \circ \alpha$ is the closure associated with A , such that $\varphi(C)$ is a complete lattice isomorphic to A . Precision of an abstract interpretation is typically defined in terms of completeness. Depending on where we compare the concrete and the abstract computations we obtain two different notions of completeness [14, 13]. If we compare the results in the abstract domain, we obtain what is called backward completeness (\mathcal{B} -completeness), while, if we compare the results in the concrete domain we obtain the so called forward completeness (\mathcal{F} -completeness). Formally, if $f : C \rightarrow C$ and $\rho \in uco(C)$, then ρ is \mathcal{B} -complete for f if $\rho \circ f = \rho \circ f \circ \rho$, while it is \mathcal{F} -complete for f if $f \circ \rho = \rho \circ f \circ \rho$. A complete over-approximation means that no false alarms are returned by the analysis, i.e., in \mathcal{B} -completeness the approximate semantics computed by manipulating abstract objects corresponds precisely

to the abstraction of the concrete semantics, while in F-completeness the concrete semantics does not lose precision by computing on abstract objects. The least fixpoint (lfp) of an operator F on a poset $\langle P, \leq \rangle$, when it exists, is denoted by $\text{lfp}^{\leq} F$, or by $\text{lfp} F$ when \leq is clear. Any continuous operator $F : C \rightarrow C$ on a complete lattice $C = \langle C, \leq_C, \vee_C, \wedge_C, \top_C, \perp_C \rangle$ admits a lfp: $\text{lfp}^{\leq_C} F = \bigvee_{n \in \mathbb{N}} F^n(\perp_C)$, where for any $i \in \mathbb{N}$ and $x \in C$: $F^0(x) = x$; $F^{i+1}(x) = F(F^i(x))$. If $F^\sharp : A \rightarrow A$ is a correct approximation of $F : C \rightarrow C$ on $\langle A, \leq_A \rangle$, then $\alpha(\text{lfp}^{\leq_C} F) \leq_A \text{lfp}^{\leq_A} F^\sharp$. Convergence can be ensured through *widening* iterations along increasing chains [6]. A widening operator $\nabla : P \times P \rightarrow P$ approximates the lub, i.e., $\forall X, Y \in P : X \leq_P (X \nabla Y)$ and $Y \leq_P (X \nabla Y)$, and it is such that the increasing chain W^i , where $W^0 = \perp$ and $W^{i+1} = W^i \nabla F(W^i)$ is not strictly increasing for \leq_P . The limit of the sequence W^i provides an upper fixpoint approximation of F on P , i.e., $\text{lfp}^{\leq_P} F \leq_P \lim_{i \rightarrow \infty} W^i$.

2.2.1 Inducing forward completeness

In [18] the authors introduced a method for inducing \mathcal{F} -completeness. So it is always possible to minimally transform a given semantics function f in order to satisfy completeness. Minimally means to find the closest function, by reducing or increasing the images of f , w.r.t. a given property we want to hold for f (in this context, completeness). Here we take in account only the case of increasing a given function, so we move upwards. For any $f : C \xrightarrow{m} C$ and $\eta, \rho \in \text{uco}(C)$ we can define

$$F_{\eta, \rho} \stackrel{\text{def}}{=} \lambda f. \lambda x. \begin{cases} \rho \circ f(x) & \text{if } x \in \eta(C) \\ f(x) & \text{otherwise} \end{cases}$$

We have that $F_{\eta, \rho}(f) = \bigsqcap \{h : C \rightarrow C \mid f \sqsubseteq h \wedge h \circ \eta = \rho \circ h \circ \eta\}$. The function $F_{\eta, \rho}(f)$ may lack monotonicity. But any function can be transformed to the closest monotone function by considering the following basic transformer:

$$M \stackrel{\text{def}}{=} \lambda f. \lambda x. \bigvee_C \{f(y) \mid y \leq_C x\}$$

So we can define the forward completeness transformer as $\mathbb{F}_{\eta, \rho} \stackrel{\text{def}}{=} M \circ F_{\eta, \rho}$ such that

$$\mathbb{F}_{\eta, \rho}(f) = \bigsqcap \{h : C \xrightarrow{m} C \mid f \sqsubseteq h \wedge h \circ \eta = \rho \circ h \circ \eta\}$$

2.3 Programming language and semantics

In this section we introduce a simple imperative programming language which is used in the rest of the work. It is a simple extension of the one introduced in [9], the main difference being the ability of programs to interact with the user, namely the programs can deal with input values.

Syntactic categories

- $n \in \text{Int}$ ▷ integers
- $b \in \text{Bool}$ ▷ booleans
- $X \in \text{Var}$ ▷ program variables
- $L \in \text{Lab}$ ▷ program labels
- $E \in \text{Exp}$ ▷ arithmetic expressions
- $B \in \text{Bexp}$ ▷ boolean expressions
- $A \in \text{Act}$ ▷ program actions
- $C \in \text{Com}$ ▷ commands
- $P \in \text{IMP} \stackrel{\text{def}}{=} \wp(\text{Com})$ ▷ programs

Value domains

- $n \in \mathbb{Z}$ ▷ integer numbers
- $b \in \mathbb{B} \stackrel{\text{def}}{=} \{\text{tt}, \text{ff}\}$ ▷ truth values
- $\rho \in \text{Env} \stackrel{\text{def}}{=} \wp(\text{Var}) \longrightarrow \wp(\mathbb{Z}_\perp)$ ▷ environments
- $\iota \in \text{Sin} \stackrel{\text{def}}{=} \mathbb{Z}^* = \bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$ ▷ standard inputs
- $\zeta \in \text{Con} \stackrel{\text{def}}{=} \text{Env} \times \text{Sin}$ ▷ contexts
- $\varsigma \in \Sigma \stackrel{\text{def}}{=} \text{Com} \times \text{Con}$ ▷ program states

Abstract syntax

- $E ::= n \mid X \mid E_1 \text{ op } E_2$ where $\text{op} \in \{+, \cdot, -, /, \dots\}$
- $B ::= b \mid E_1 < E_2 \mid E_1 = E_2 \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B$
- $A ::= X := E \mid \text{input } X \mid \text{skip}$
- $C ::= L : A \rightarrow L' \mid L : B \rightarrow \{L_{\text{tt}}, L_{\text{ff}}\} \mid L : \text{stop};$

Note that the command $L : \text{stop};$ can be simulated by $L : \text{skip} \rightarrow \perp;$ and the command $L : \text{skip} \rightarrow L';$ can be simulated by $L : \text{tt} \rightarrow \{L', \perp\};$ where \perp is the undefined label. In the follow, with a little abuse of notation, we refer with \perp every undefined value, of any type. For a given set S , we indicate with S_\perp the set S augmented with the undefined symbol, so $S_\perp \stackrel{\text{def}}{=} S \cup \{\perp\}.$

Auxiliary functions

- Labels:

- $lab \llbracket L : A \rightarrow L'; \rrbracket \stackrel{\text{def}}{=} L$
- $lab \llbracket L : B \rightarrow \{L_{tt}, L_{ff}\}; \rrbracket \stackrel{\text{def}}{=} L$
- $lab \llbracket L : \text{stop}; \rrbracket \stackrel{\text{def}}{=} L$
- $lab \llbracket P \rrbracket \stackrel{\text{def}}{=} \{lab \llbracket C \rrbracket \mid C \in P\}$

- Variables:

- $var \llbracket D \rrbracket \stackrel{\text{def}}{=} \{X \in \mathbf{Var} \mid X \text{ is in } D\}$ where $D \in \{E, B\}$
- $var \llbracket X := E \rrbracket \stackrel{\text{def}}{=} \{X\} \cup var \llbracket E \rrbracket$
- $var \llbracket \text{input } X \rrbracket \stackrel{\text{def}}{=} \{X\}$
- $var \llbracket \text{skip} \rrbracket \stackrel{\text{def}}{=} \emptyset$
- $var \llbracket L : A \rightarrow L'; \rrbracket \stackrel{\text{def}}{=} var \llbracket A \rrbracket$
- $var \llbracket L : B \rightarrow \{L_{tt}, L_{ff}\}; \rrbracket \stackrel{\text{def}}{=} var \llbracket B \rrbracket$
- $var \llbracket L : \text{stop}; \rrbracket \stackrel{\text{def}}{=} \emptyset$
- $var \llbracket P \rrbracket \stackrel{\text{def}}{=} \bigcup_{C \in P} var \llbracket C \rrbracket$

- Actions of a command:

- $act \llbracket L : A \rightarrow L'; \rrbracket \stackrel{\text{def}}{=} A$
- $act \llbracket L : B \rightarrow \{L_{tt}, L_{ff}\}; \rrbracket \stackrel{\text{def}}{=} \emptyset$
- $act \llbracket L : \text{stop}; \rrbracket \stackrel{\text{def}}{=} \emptyset$

- Successors of a commands:

- $suc \llbracket L : A \rightarrow L'; \rrbracket \stackrel{\text{def}}{=} \{L'\}$
- $suc \llbracket L : B \rightarrow \{L_{tt}, L_{ff}\}; \rrbracket \stackrel{\text{def}}{=} \{L_{tt}, L_{ff}\}$
- $suc \llbracket L : \text{stop}; \rrbracket \stackrel{\text{def}}{=} \{\perp\}$
- $suc \llbracket P \rrbracket \stackrel{\text{def}}{=} \bigcup_{C \in P} suc \llbracket C \rrbracket$

- Environments:

- $env \llbracket \mathcal{X} \rrbracket \stackrel{\text{def}}{=} \{\rho \in \mathbf{Env} \mid \text{dom}(\rho) = \mathcal{X}\}$ where $\mathcal{X} \subseteq \mathbf{Var}$
- $env \llbracket P \rrbracket \stackrel{\text{def}}{=} env \llbracket var \llbracket P \rrbracket \rrbracket$

- Contexts:

- $con \llbracket \rho \rrbracket \stackrel{\text{def}}{=} \{\langle \rho, \iota \rangle \in \mathbf{Con} \mid \iota \in \mathbf{Sin}\}$ where $\rho \in \mathbf{Env}$

$$- \text{con} \llbracket P \rrbracket \stackrel{\text{def}}{=} \{ \text{con} \llbracket \rho \rrbracket \mid \rho \in \text{env} \llbracket P \rrbracket \}$$

Furthermore we have two functions that deal with the standard inputs. The first is $\text{top} : \text{Sin} \rightarrow \mathbb{Z}_\perp$ which, given a standard input, returns the next value that have to be passed to the program, namely

$$\text{top}(\iota) \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } \iota = \epsilon \\ z \in \mathbb{Z} & \text{if } \iota = z\iota' \end{cases}$$

The second is $\text{next} : \text{Sin} \rightarrow \text{Sin}$ which, given a standard input, returns another standard input without the current value which have to be passed to the program, namely

$$\text{next}(\iota) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } \iota = \epsilon \\ \iota' \in \text{Sin} & \text{if } \iota = z\iota' \wedge z \in \mathbb{Z} \end{cases}$$

Semantics

Arithmetic expressions $\mathcal{E} \llbracket E \rrbracket \in \text{Con} \rightarrow \mathbb{Z}_\perp$

- $\mathcal{E} \llbracket n \rrbracket \zeta \stackrel{\text{def}}{=} n$
- $\mathcal{E} \llbracket X \rrbracket \langle \rho, \iota \rangle \stackrel{\text{def}}{=} \rho(X)$
- $\mathcal{E} \llbracket E_1 \text{ op } E_2 \rrbracket \zeta \stackrel{\text{def}}{=} \begin{cases} \mathcal{E} \llbracket E_1 \rrbracket \zeta \text{ op } \mathcal{E} \llbracket E_2 \rrbracket \zeta & \text{if } \mathcal{E} \llbracket E_1 \rrbracket \zeta \in \mathbb{Z} \text{ and } \mathcal{E} \llbracket E_2 \rrbracket \zeta \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$

Boolean expressions $\mathcal{B} \llbracket B \rrbracket \in \text{Con} \rightarrow \mathbb{B}_\perp$

- $\mathcal{B} \llbracket \text{tt} \rrbracket \zeta \stackrel{\text{def}}{=} \text{tt}$
- $\mathcal{B} \llbracket \text{ff} \rrbracket \zeta \stackrel{\text{def}}{=} \text{ff}$
- $\mathcal{B} \llbracket E_1 < E_2 \rrbracket \zeta \stackrel{\text{def}}{=} \begin{cases} \mathcal{E} \llbracket E_1 \rrbracket \zeta < \mathcal{E} \llbracket E_2 \rrbracket \zeta & \text{if } \mathcal{E} \llbracket E_1 \rrbracket \zeta \in \mathbb{Z} \text{ and } \mathcal{E} \llbracket E_2 \rrbracket \zeta \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$
- $\mathcal{B} \llbracket E_1 = E_2 \rrbracket \zeta \stackrel{\text{def}}{=} \begin{cases} \mathcal{E} \llbracket E_1 \rrbracket \zeta = \mathcal{E} \llbracket E_2 \rrbracket \zeta & \text{if } \mathcal{E} \llbracket E_1 \rrbracket \zeta \in \mathbb{Z} \text{ and } \mathcal{E} \llbracket E_2 \rrbracket \zeta \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$
- $\mathcal{B} \llbracket \neg B \rrbracket \zeta \stackrel{\text{def}}{=} \begin{cases} \neg \mathcal{B} \llbracket B \rrbracket \zeta & \text{if } \mathcal{B} \llbracket B \rrbracket \zeta \in \mathbb{B} \\ \perp & \text{otherwise} \end{cases}$
- $\mathcal{B} \llbracket B_1 \wedge B_2 \rrbracket \zeta \stackrel{\text{def}}{=} \begin{cases} \mathcal{B} \llbracket B_1 \rrbracket \zeta \wedge \mathcal{B} \llbracket B_2 \rrbracket \zeta & \text{if } \mathcal{B} \llbracket B_1 \rrbracket \zeta \in \mathbb{B} \text{ and } \mathcal{B} \llbracket B_2 \rrbracket \zeta \in \mathbb{B} \\ \perp & \text{otherwise} \end{cases}$
- $\mathcal{B} \llbracket B_1 \vee B_2 \rrbracket \zeta \stackrel{\text{def}}{=} \begin{cases} \mathcal{B} \llbracket B_1 \rrbracket \zeta \vee \mathcal{B} \llbracket B_2 \rrbracket \zeta & \text{if } \mathcal{B} \llbracket B_1 \rrbracket \zeta \in \mathbb{B} \text{ and } \mathcal{B} \llbracket B_2 \rrbracket \zeta \in \mathbb{B} \\ \perp & \text{otherwise} \end{cases}$

Actions $\mathcal{A}[[A]] \in \text{Con} \longrightarrow \text{Con}$

- $\mathcal{A}[[X := E]]\langle \rho, \iota \rangle \stackrel{\text{def}}{=} \langle \rho[X \leftarrow \mathcal{E}[[E]]\langle \rho, \iota \rangle], \iota \rangle$
- $\mathcal{A}[[\text{input } X]]\langle \rho, \iota \rangle \stackrel{\text{def}}{=} \begin{cases} \langle \rho[X \leftarrow \text{top}(\iota)], \text{next}(\iota) \rangle & \text{if } \text{top}(\iota) \neq \perp \\ \langle \rho[X \leftarrow \perp], \iota \rangle & \text{otherwise} \end{cases}$
- $\mathcal{A}[[\text{skip}]]\zeta \stackrel{\text{def}}{=} \zeta$

Commands $\mathcal{C}[[C]] \in \text{Con} \longrightarrow \wp(\text{Com} \times \text{Con})$

- $\mathcal{C}[[L : A \rightarrow L';]]\zeta \stackrel{\text{def}}{=} \{ \langle C, \zeta' \rangle \mid \text{lab}[[C]] = L' \wedge \zeta' = \mathcal{A}[[A]]\zeta \}$
- $\mathcal{C}[[L : B \rightarrow \{L_{\text{tt}}, L_{\text{ff}}\};]]\zeta \stackrel{\text{def}}{=} \left\{ \langle C, \zeta \rangle \mid \text{lab}[[C]] = \begin{cases} L_{\text{tt}} & \text{if } \mathcal{B}[[B]]\zeta = \text{tt} \\ L_{\text{ff}} & \text{if } \mathcal{B}[[B]]\zeta = \text{ff} \end{cases} \right\}$
- $\mathcal{C}[[L : \text{stop};]]\zeta \stackrel{\text{def}}{=} \emptyset$

2.3.1 Semantics of programs

The *transition relation* $\mathbf{S} \in \Sigma \longrightarrow \wp(\Sigma)$ specifies the successor states of a given state:

$$\mathbf{S}(\langle C, \zeta \rangle) \stackrel{\text{def}}{=} \{ \langle C', \zeta' \rangle \mid \zeta' = \mathcal{A}[[\text{act}[[C]]]]\zeta \wedge \text{lab}[[C']] \in \text{succ}[[C]] \}$$

The set of states of a program is defined as:

$$\text{sts}[[P]] \stackrel{\text{def}}{=} P \times \text{con}[[P]]$$

The *transitional semantics* $\mathbf{S}[[P]] \in \text{sts}[[P]] \longrightarrow \wp(\text{sts}[[P]])$ of a program P is:

$$\mathbf{S}[[P]]\langle C, \zeta \rangle \stackrel{\text{def}}{=} \{ \langle C', \zeta' \rangle \in \mathbf{S}(\langle C, \zeta \rangle) \mid \zeta, \zeta' \in \text{con}[[P]] \wedge C, C' \in P \}$$

A *trace* $\sigma \in \Sigma$ is a sequence of states $\varsigma_0, \dots, \varsigma_{n-1}$ of length $|\sigma| = n > 0$ such that for all $i \in [1, n)$ we have $\varsigma_i \in \mathbf{S}(\varsigma_{i-1})$. With Σ^+ we indicate the set of all *finite traces*. If σ is a finite trace, we indicate with σ_{\vdash} its first element, i.e. $\sigma_{\vdash} = \sigma_0$, and we indicate with σ_{\dashv} its last element, i.e. $\sigma_{\dashv} = \sigma_{|\sigma|-1}$.

The *partial finite traces semantics* $(P)_{\oplus} \subseteq \Sigma^+$ of a program P is the least fix-point of the so called *transition function* $F_P^{\oplus} \in \wp(\Sigma^+) \xrightarrow{m} \wp(\Sigma^+)$ defined as:

$$F_P^{\oplus} \stackrel{\text{def}}{=} \lambda S. \text{sts}[[P]] \cup \{ \varsigma \zeta' \sigma \mid \zeta' \in \mathbf{S}[[P]](\varsigma) \wedge \zeta' \sigma \in S \}$$

So $(P)_{\oplus} = \text{lfp}_{\subseteq}^{\text{C}} F_P^{\oplus}$. If we are only interested in those executions of a program P starting from a given set $L^P \subseteq \text{lab}[[P]]$ of entry points, so that $I^P \stackrel{\text{def}}{=} \{ \langle C, \zeta \rangle \mid \zeta \in \text{con}[[P]] \wedge C \in P \wedge \text{lab}[[C]] \in L^P \}$ is the set of initial states, we can

consider the partial traces semantics $\langle P \rangle_{\oplus}^{I^P} \subseteq \Sigma^+$ of P which is the set of partial traces $\sigma \in \Sigma^+$ starting from an initial state $\varsigma_0 \in I^P$. The partial traces semantics $\langle P \rangle_{\oplus}^{I^P}$ can be expressed in fixpoint form as $\text{lfp}_{\emptyset}^{\subseteq} F\iota_P^{\oplus}(I^P)$ where $F\iota_P^{\oplus}(I^P) \in \wp(\Sigma^+) \xrightarrow{m} \wp(\Sigma^+)$ is:

$$F\iota_P^{\oplus}(I^P) \stackrel{\text{def}}{=} \lambda S . I^P \cup \{ \sigma \varsigma \varsigma' \mid \sigma \varsigma \in S \wedge \varsigma' \in \mathbf{S}[[P]](\varsigma) \}$$

So we can define a function, called *partial input semantics*, $\langle P \rangle_{\oplus} \in \wp(\Sigma) \rightarrow \wp(\Sigma^+)$ defined as follow:

$$\langle P \rangle_{\oplus} \stackrel{\text{def}}{=} \lambda S . \text{lfp}_{\emptyset}^{\subseteq} F\iota_P^{\oplus}(S) = \lambda S . \langle P \rangle_{\oplus}^S$$

A state ς is *blocking* (or *final*), w.r.t. a program P , if $\mathbf{S}[[P]](\varsigma) = \emptyset$. So the set of blocking states of the program P is $T^P \stackrel{\text{def}}{=} \{ \langle C, \varsigma \rangle \in \text{sts}[[P] \mid \text{succ}[[C]] \not\subseteq \text{lab}[[P]] \}$. A *maximal finite trace* of a program P , is a trace $\sigma \in \Sigma^+$ of length n where the last state σ_{n-1} is blocking. $\langle P \rangle^n$ is the set of all the finite traces of length n of the program P . The *maximal finite traces semantics* $\langle P \rangle_+$ of the program P is given by the union of all maximal finite traces of length $n > 0$, namely $\langle P \rangle_+ \stackrel{\text{def}}{=} \bigcup_{n>0} \{ \sigma \in \langle P \rangle^n \mid \sigma_+ \in T^P \}$. This semantics can be expressed as the least fixpoint of the transition function $F_P^+ \in \wp(\Sigma^+) \xrightarrow{m} \wp(\Sigma^+)$ defined as follow:

$$F_P^+ \stackrel{\text{def}}{=} \lambda S . T^P \cup \{ \varsigma \varsigma' \sigma \mid \varsigma' \in \mathbf{S}[[P]](\varsigma) \wedge \varsigma' \sigma \in S \}$$

Similarly, we can define a function, called *maximal input semantics*, $\langle P \rangle_+ \in \wp(\Sigma) \rightarrow \wp(\Sigma^+)$ defined as follow:

$$\langle P \rangle_+ \stackrel{\text{def}}{=} \lambda S . \{ \sigma \in \langle P \rangle_+ \mid \sigma_0 \in S \}$$

Unfortunately, it seems that this semantic function can't be expressed in fixpoint form in the lattice $\langle \wp(\Sigma^+), \subseteq, \cup, \cap, \Sigma^+, \emptyset \rangle$. A tricky way for avoiding the problem is to calculate the function as the combination of partial input and maximal traces semantics, so as:

$$\langle P \rangle_+ = \lambda S . \langle P \rangle_{\oplus}(S) \cap \langle P \rangle_+$$

But a better solution would be the definition of a specific semantic domain in which we are able to compute this function directly.

Prefix ordering Let $\text{pref} : \Sigma^+ \rightarrow \wp(\Sigma^+)$ be a function that returns set of prefix of a given trace, so $\text{pref}(\sigma) \stackrel{\text{def}}{=} \{ \sigma' \in \Sigma^+ \mid \exists \sigma'' \in \Sigma^+ . \sigma = \sigma' \sigma'' \}$. We can define the relation $\leq \subseteq \wp(\Sigma^+) \times \wp(\Sigma^+)$ which is a partial order between sets of traces:

$$X \leq Y \Leftrightarrow \begin{aligned} & \forall \sigma \in X \exists \sigma' \in Y . \sigma \in \text{pref}(\sigma') \wedge \\ & (\forall \sigma' \in Y \exists \sigma \in X . \sigma \in \text{pref}(\sigma') \Rightarrow Y \subseteq X) \end{aligned}$$

Reflexivity.

$\forall X \in \wp(\Sigma^+)$. $X \leq X$ holds, in fact $\forall \sigma \in X \exists \sigma' \in X . \sigma \in \text{pref}(\sigma')$ holds for $\sigma' = \sigma$ and $X \subseteq X$ is trivially true.

Antisymmetry.

$\forall X, Y \in \wp(\Sigma^+)$:

- b** from $X \leq Y$ it follows that $\forall \sigma \in X \exists \sigma' \in Y . \sigma \in \text{pref}(\sigma')$ and so from $Y \leq X$ (the second part of the conjunction) it follows that $X \subseteq Y$
- a** from $Y \leq X$ it follows that $\forall \sigma' \in Y \exists \sigma \in X . \sigma' \in \text{pref}(\sigma)$ and so from $X \leq Y$ (the second part of the conjunction) it follows that $Y \subseteq X$

From **a** and **b** it follows that $X = Y$.

Transitivity.

$\forall X, Y, Z \in \wp(\Sigma^+)$:

- b** from $X \leq Y$ it follows that $\forall \sigma^x \in X \exists \sigma^y \in Y . \sigma^x \in \text{pref}(\sigma^y)$
- a** from $Y \leq Z$ it follows that $\forall \sigma^{y'} \in Y \exists \sigma^z \in Z . \sigma^{y'} \in \text{pref}(\sigma^z)$

Considering **a** and **b** it's easy to find for all $\sigma^x \in X$ a $\sigma^z \in Z$ such that $\sigma^x \in \text{pref}(\sigma^z)$, in fact with $\sigma^y = \sigma^{y'}$ the relation is satisfied. In the end from $Z \subseteq Y$ and $Y \subseteq X$ it follows that $Z \subseteq X$. So $X \leq Z$ holds.

Now we have to define the elements which extend the poset $(\wp(\Sigma^+), \leq)$. The *least upper bound* \uplus is defined as:

$$\uplus \mathcal{X} \stackrel{\text{def}}{=} \left\{ \sigma \in \bigcup_{X \in \mathcal{X}} X \mid \forall \sigma' \in \bigcup_{X \in \mathcal{X}} X . \sigma \in \text{pref}(\sigma') \Rightarrow \sigma = \sigma' \right\}$$

Clearly, for all $X \in \mathcal{X} \subseteq \wp(\Sigma^+)$ we have that $X \leq \uplus \mathcal{X}$, namely $\uplus \mathcal{X}$ is an upper bound of \mathcal{X} , but we have to prove that it is the least. In order to prove that $\forall \mathcal{X} \subseteq \wp(\Sigma^+) \forall \varpi \in \wp(\Sigma^+) . (\forall X \in \mathcal{X} . X \leq \varpi) \Rightarrow \uplus \mathcal{X} \leq \varpi$ we show that its negate is false, i.e. we prove that it doesn't exist $\varpi \in \wp(\Sigma^+)$ such that $(\forall X \in \mathcal{X} . X \leq \varpi) \Rightarrow (\varpi \leq \uplus \mathcal{X} \wedge \varpi \neq \uplus \mathcal{X})$.

$\forall X \in \mathcal{X} . X \leq \varpi$ means that $\forall \sigma \in \bigcup_{X \in \mathcal{X}} X \exists \sigma' \in \varpi . \sigma \in \text{pref}(\sigma')$ and $\varpi \leq \uplus \mathcal{X}$ means that $\forall \sigma' \in \varpi \exists \sigma'' \in \uplus \mathcal{X} . \sigma' \in \text{pref}(\sigma'')$. By definition, $\uplus \mathcal{X} \subseteq \bigcup_{X \in \mathcal{X}} X$ so $\forall \sigma' \in \varpi \exists \sigma'' \in \bigcup_{X \in \mathcal{X}} X . \sigma' \in \text{pref}(\sigma'')$. But from the last proposition and from $\forall \sigma \in \bigcup_{X \in \mathcal{X}} X \exists \sigma' \in \varpi . \sigma \in \text{pref}(\sigma')$ it follows that $\varpi = \bigcup_{X \in \mathcal{X}} X$ (and so $\uplus \mathcal{X} \subseteq \varpi$). By the fact that $\uplus \mathcal{X} \neq \varpi$ we have that $\exists \sigma'' \in \uplus \mathcal{X} \forall \sigma' \in \varpi . \sigma'' \notin \text{pref}(\sigma')$ which is absurd because $\uplus \mathcal{X} \subseteq \varpi$.

The *bottom* element is $\emptyset \in \wp(\Sigma^+)$, i.e. $\forall X \in \wp(\Sigma^+) . \emptyset \leq X$ holds. In fact, with no traces, the conditions of the relation are vacuously true and, if $X = \emptyset$, $\emptyset \leq \emptyset$ trivially holds.

Now we have all the ingredients. In fact for all $\mathcal{X} \subseteq \wp(\Sigma^+)$ it's easy to note that $\uplus \mathcal{X}$ exists and it is a set of finite traces, so $\uplus \mathcal{X} \in \wp(\Sigma^+)$. So, by the fact that $(\wp(\Sigma^+), \leq)$ has an infimum (bottom), in addition to the fact that

for each subset of $\wp(\Sigma^+)$ there is a least upper bound in $\wp(\Sigma^+)$, we get that $\langle \wp(\Sigma^+), \leq, \uplus, \emptyset \rangle$ is a directed-complete partial order (CPO or DCPO).

Finally let's define the maximal input semantic in this new domain. Like for partial traces semantics, we can consider the maximal traces semantics $\langle P \rangle_+^{I^P} \subseteq \Sigma^+$ of P which is the set of maximal traces $\sigma \in \Sigma^+$ starting from an initial state $\varsigma_0 \in I^P$. Recall that the maximal finite traces semantics $\langle P \rangle_+$ is equal to $\bigcup_{n>0} \{\sigma \in \langle P \rangle^n \mid \sigma_{\downarrow} \in T^P\}$, so this semantics computed starting from states in I^P is defined as $\langle P \rangle_+^{I^P} \stackrel{\text{def}}{=} \bigcup_{n>0} \{\sigma \in \langle P \rangle^n \mid \sigma_{\uparrow} \in I^P \wedge \sigma_{\downarrow} \in T^P\}$. In the follow we indicate with $\langle P \rangle_{I^P}^n$ the set $\{\sigma \in \langle P \rangle^n \mid \sigma_{\uparrow} \in I^P\}$ and with $\langle P \rangle_{I^P}^{\bar{n}}$ the set $\{\sigma \in \langle P \rangle^n \mid \sigma_{\uparrow} \in I^P \wedge \sigma_{\downarrow} \in T^P\}$, so $\langle P \rangle_+^{I^P}$ can be rewritten as $\bigcup_{n>0} \langle P \rangle_{I^P}^{\bar{n}}$. Now we can note that the following holds:

$$\bigcup_{n>0} \langle P \rangle_{I^P}^{\bar{n}} = \biguplus_{n>0} \langle P \rangle_{I^P}^{\bar{n}}$$

because for every n the traces in $\langle P \rangle_{I^P}^{\bar{n}}$ are not prefixes of any trace in $\langle P \rangle_{I^P}^{\bar{n}}$, due to the fact that they are maximal (and so the lub returns the union of this two sets).

So the semantics can be expressed as $\text{lfp}_{\emptyset}^{\leq} F_{I^P}^+(I^P)$ of the Scott-continuous, and so monotone, function $F_{I^P}^+(I^P) \in \wp(\Sigma^+) \xrightarrow{c} \wp(\Sigma^+)$:

$$F_{I^P}^+(I^P) \stackrel{\text{def}}{=} \lambda S. \langle P \rangle_{I^P}^1 \uplus \{\sigma\varsigma\varsigma' \mid \sigma\varsigma \in S \wedge \varsigma' \in \mathbf{S}[[P]](\varsigma)\}$$

The first iterates of F_{P,I^P}^+ for $\text{lfp}_{\emptyset}^{\leq} F_{P,I^P}^+$, where $F_{P,I^P}^+ = F_{I^P}^+(I^P)$, are:

$$\begin{aligned} X^0 &= \emptyset \\ X^1 &= F_{P,I^P}^+(X^0) = \langle P \rangle_{I^P}^1 \uplus \{\sigma\varsigma\varsigma' \mid \sigma\varsigma \in \emptyset \wedge \varsigma' \in \mathbf{S}[[P]](\varsigma)\} = I^P = \langle P \rangle_{I^P}^1 \\ X^2 &= F_{P,I^P}^+(X^1) = \langle P \rangle_{I^P}^1 \uplus \{\sigma\varsigma\varsigma' \mid \sigma\varsigma \in X^1 \wedge \varsigma' \in \mathbf{S}[[P]](\varsigma)\} = \\ &= \langle P \rangle_{I^P}^1 \uplus (\langle P \rangle_{I^P}^{\bar{1}} \uplus \langle P \rangle_{I^P}^2) = \langle P \rangle_{I^P}^{\bar{1}} \uplus \langle P \rangle_{I^P}^2 \\ X^3 &= F_{P,I^P}^+(X^2) = \langle P \rangle_{I^P}^1 \uplus \{\sigma\varsigma\varsigma' \mid \sigma\varsigma \in X^2 \wedge \varsigma' \in \mathbf{S}[[P]](\varsigma)\} = \\ &= \langle P \rangle_{I^P}^1 \uplus (\langle P \rangle_{I^P}^{\bar{1}} \uplus \langle P \rangle_{I^P}^{\bar{2}} \uplus \langle P \rangle_{I^P}^3) = \langle P \rangle_{I^P}^{\bar{1}} \uplus \langle P \rangle_{I^P}^{\bar{2}} \uplus \langle P \rangle_{I^P}^3 \end{aligned}$$

By recurrence the n -th iterate of F_{P,I^P}^+ is:

$$X^n = \biguplus_{i=1}^{n-1} \langle P \rangle_{I^P}^{\bar{i}} \uplus \langle P \rangle_{I^P}^n$$

In fact

$$\begin{aligned} F_{P,I^P}^+(X^n) &= \langle P \rangle_{I^P}^1 \uplus \{\sigma\varsigma\varsigma' \mid \sigma\varsigma \in X^n \wedge \varsigma' \in \mathbf{S}[[P]](\varsigma)\} = \\ &= \langle P \rangle_{I^P}^1 \uplus (\biguplus_{i=1}^n \langle P \rangle_{I^P}^{\bar{i}} \uplus \langle P \rangle_{I^P}^{n+1}) = \\ &= \biguplus_{i=1}^n \langle P \rangle_{I^P}^{\bar{i}} \uplus \langle P \rangle_{I^P}^{n+1} = X^{n+1} \end{aligned}$$

Due to the fact that $F_{P,IP}^+$ is Scott-continuous and that it's defined over a DCPO, $F_{P,IP}^+$ admits a fixpoint, i.e. exists $k \in \mathbb{N}$ such that $F_{P,IP}^+(X^k) = X^k$, and it is exactly the least upper bound of the Kleene chain of $F_{P,IP}^+$ starting from \emptyset . Due this considerations we get that

$$F_{P,IP}^+(X^k) = \bigsqcup_{i=1}^k \langle P \rangle_{IP}^{\bar{i}} \sqcup \langle P \rangle_{IP}^{k+1} = \bigsqcup_{i=1}^{k-1} \langle P \rangle_{IP}^{\bar{i}} \sqcup \langle P \rangle_{IP}^k = X^k$$

and so that

$$\begin{aligned} \langle P \rangle_{IP}^{\bar{k}} \cup \langle P \rangle_{IP}^{k+1} &= \langle P \rangle_{IP}^k \\ \langle P \rangle_{IP}^{\bar{k}} \cup \langle P \rangle_{IP}^{k+1} &= \langle P \rangle_{IP}^{\bar{k}} \cup \{\sigma \in \langle P \rangle_{IP}^k \mid \sigma_{\rightarrow} \notin T^P\} \\ \langle P \rangle_{IP}^{k+1} &= \{\sigma \in \langle P \rangle_{IP}^k \mid \sigma_{\rightarrow} \notin T^P\} \end{aligned}$$

The only way to make the last equation true is to have $\langle P \rangle_{IP}^{k+1} = \emptyset$ and $\{\sigma \in \langle P \rangle_{IP}^k \mid \sigma_{\rightarrow} \notin T^P\} = \emptyset$, due to the fact that $\{\sigma \in \langle P \rangle_{IP}^k \mid \sigma_{\rightarrow} \notin T^P\}$ and $\langle P \rangle_{IP}^{k+1}$ don't share any element. So finally we have that exists $k \in \mathbb{N}$ such that $F_{P,IP}^+(X^k) = \bigsqcup_{n>0} \langle P \rangle_{IP}^{\bar{n}}$

Then we can define the maximal input semantics as:

$$\langle P \rangle_+ \stackrel{\text{def}}{=} \lambda S. \text{lf}p_{\emptyset}^{\leq} F \iota_P^+(S) = \lambda S. \langle P \rangle_+^S$$

2.3.2 Abstract semantics

If we are interested in the abstract semantics of a program, i.e. given a program we need its abstract interpretation in a specific abstract domain, then we can obtain a sound abstract semantics just only calculating the semantics using the *best correct approximation* of the transfer function instead the transfer function itself. So if a semantics is calculated in a concrete domain C as fixpoint of the transfer function F then we can calculate its abstract interpretation in an abstract domain $\rho \in \text{uco}(C)$ as fixpoint of the function $\rho F \rho$. If the abstract domain is \mathcal{B} -complete for F then we have that $\text{lf}p \rho F \rho = \rho(\text{lf}p F)$. For example, let $\langle \wp(\Sigma^+), \subseteq, \cup, \cap, \Sigma^+, \emptyset \rangle$ the concrete domain and $\langle P \rangle_{\oplus}$ the semantics calculated as fixpoint of the transfer function F_P^{\oplus} . Then the best correct approximation of P in ρ is:

$$\langle P \rangle_{\oplus}^{\rho} \stackrel{\text{def}}{=} \text{lf}p_{\emptyset}^{\subseteq} \rho \circ F_P^{\oplus} \circ \rho$$

Let $\langle \wp(\Sigma^+), \leq, \sqcup, \emptyset \rangle$ the concrete domain and $\langle P \rangle_+^S$ the semantics calculated as fixpoint of the transfer function $F \iota_P^+(S)$. Then the best correct approximation of P in ρ is:

$$\langle P \rangle_+^{S\rho} \stackrel{\text{def}}{=} \text{lf}p_{\emptyset}^{\leq} \rho \circ F \iota_P^+(S) \circ \rho$$

So the we can define the abstract maximal input semantics of P in ρ as:

$$\langle P \rangle_+^{\rho} \stackrel{\text{def}}{=} \lambda S. \text{lf}p_{\emptyset}^{\leq} \rho \circ F \iota_P^+(S) \circ \rho$$

2.4 Abstract non-interference

Abstract non-interference (ANI) [16] is a weakening of non-interference by abstract interpretation. Let $\eta, \rho \in uco(\wp(\mathbb{Z}_\perp^L))$ and $\phi \in uco(\wp(\mathbb{Z}_\perp^H))$, where \mathbb{Z}_\perp^L and \mathbb{Z}_\perp^H are the domains of public (L) and private (H) variables. η and ρ characterize the attacker, instead ϕ states what, of the private data, can flow to the output observation, the so called declassification of ϕ [19]. A program P satisfies ANI, and we write $[\eta]P(\phi \Rightarrow \rho)$, if $\forall h_1, h_2 \in \mathbb{Z}_\perp^H$ and $\forall l_1, l_2 \in \mathbb{Z}_\perp^L$:

$$\eta(l_1) = \eta(l_2) \wedge \phi(h_1) = \phi(h_2) \Rightarrow \rho(\llbracket P \rrbracket_{Den}(\langle h_1, l_1 \rangle)^L) = \rho(\llbracket P \rrbracket_{Den}(\langle h_2, l_2 \rangle)^L)$$

Where with $\llbracket P \rrbracket_{Den} \in \wp(\Sigma) \rightarrow \Sigma$ we denote the angelic denotational semantics of the program P [17]. This notion says that, whenever the attacker is able to observe the input property η and the ρ property of the output, then it can observe nothing more than the property ϕ of private input. In order to model non-interference in code transformations, such as software watermarking, we consider an higher-order version of ANI, where the objects of observations are programs instead of values. Hence, we have a part of a program (semantics) that can change and that is secret, and the environment which remains the same up to an observable property: these are the new private and public inputs. The function that, in some way, has to hide the change is now a program transformer, which takes the two parts of the program and provides a program as result. The output observation is the best correct approximation of the resulting program.

Here the semantics we take in account is the maximal finite trace semantics. Let \mathcal{P} be the set of cover programs, \mathcal{Q} the set of secret program and $\mathcal{I} : \text{IMP} \times \text{IMP} \rightarrow \text{IMP}$ an integration function. As usual, the attacker is modelled as a couple $\langle \eta, \rho \rangle$, whit $\eta, \rho \in uco(\wp(\Sigma^+))$, that represents the input and output public observation power. Instead $\phi \in uco(\wp(\Sigma^+))$ is the property of the secret input.

DEFINITION 1 (HOANI_{bca} for maximal finite traces semantics).

The integration program \mathcal{I} , given $\eta, \phi, \rho \in uco(\wp(\Sigma^+))$, satisfies higher-order abstract non-interference (for maximal finite traces semantics) w.r.t $\langle \eta, \phi, \rho \rangle$ and $\langle \mathcal{P}, \mathcal{Q} \rangle$ if:

$$\forall P_1, P_2 \in \mathcal{P} \forall Q_1, Q_2 \in \mathcal{Q} . \\ \llbracket P_1 \rrbracket_+^\eta = \llbracket P_2 \rrbracket_+^\eta \wedge \llbracket Q_1 \rrbracket_+^\phi = \llbracket Q_2 \rrbracket_+^\phi \Rightarrow \llbracket \mathcal{I}(P_1, Q_1) \rrbracket_+^\rho = \llbracket \mathcal{I}(P_2, Q_2) \rrbracket_+^\rho$$

We write $\overset{H}{+}[\eta]\mathcal{I}(\phi \Rightarrow \rho)_{bca}$ to indicate that the program \mathcal{I} satisfies higher-order abstract non-interference (for maximal finite traces semantics) w.r.t $\langle \eta, \phi, \rho \rangle$.

2.4.1 Deriving attackers

In this section we introduce a method for defining attackers, via abstract interpretation, for which a program is safe. In this case security refers to abstract non-interference. In particular, it is interested to characterize the most concrete (i.e. the most precise) attacker for which a program is safe. In fact it can be shown that if it true $[\eta]P(\phi \Rightarrow \rho)$ then, for any β such that $\rho \sqsubseteq \beta$, it holds $[\eta]P(\phi \Rightarrow \beta)$ [16]. That is, if abstract non-interference holds observing in output the property ρ , then it holds also observing in output any more abstract properties (which distinguishes less elements) than ρ . In [16, 19] is shown how to derive this attacker for abstract non-interference. Following that we try to derive the attackers w.r.t. HOANI_{bca} for maximal finite traces semantics. In this case the sets of values are replaced by sets of traces. The attackers are defined by pairs of abstract domains, therefore it will be characterized a domains transformer, parametric in the program to be analyzed, which transforms each non-secret output abstraction into the nearest abstraction for which non-interference holds. We fix a public input properties η and a private input property ϕ , with $\eta, \phi \in \text{uco}(\wp(\Sigma^+))$, and we consider an abstraction $\rho \in \text{uco}(\wp(\Sigma^+))$ such that it doesn't hold ${}^H_+[\eta]\mathfrak{J}(\phi \Rightarrow \rho)_{bca}$ for a program $\mathfrak{J} \in \text{IMP}$. We can derive the most concrete $\hat{\rho}$ more abstract than ρ such that ${}^H_+[\eta]\mathfrak{J}(\phi \Rightarrow \hat{\rho})_{bca}$ holds, which is called *higher-order abstract secret kernel* if \mathfrak{J} .

DEFINITION 2 (Secret kernel for HOANI).

Let $\mathfrak{J} \in \text{IMP}$ and $\mathcal{K}_{\mathfrak{J}, \eta, (\phi)}^{\text{H}^+} \in \text{uco}(\wp(\Sigma^+)) \longrightarrow \text{uco}(\wp(\Sigma^+))$:

$$\mathcal{K}_{\mathfrak{J}, \eta, (\phi)}^{\text{H}^+} \stackrel{\text{def}}{=} \lambda \rho. \bigsqcap \{ \beta \mid \rho \sqsubseteq \beta \wedge {}^H_+[\eta]\mathfrak{J}(\phi \Rightarrow \beta)_{bca} \}$$

is the higher-order abstract secret kernel transformer for \mathfrak{J} .

To characterize this transformer we must identify how much a property is safe. Program properties, in this case, are collections of traces so we have to characterize the sets of traces which can be present in the kernel in order to make non-interference to hold. To obtain this, we define a predicate, on sets of traces, which identifies the elements of a given domain that form the secret kernel. Clearly these elements must ensure non-interference, so they must abstract in the same element all objects which must not be distinguishable. To do this we must define two equivalence relations, which bring together the programs according to a property on public programs (η) and a property on private programs (ϕ). Let $\equiv_\eta, \equiv_\phi \subseteq \text{IMP} \times \text{IMP}$ such that:

$$\begin{aligned} \equiv_\eta &\stackrel{\text{def}}{=} \{ \langle P, P' \rangle \mid P, P' \in \text{P} \wedge \langle P \rangle_+^\eta = \langle P' \rangle_+^\eta \} \\ \equiv_\phi &\stackrel{\text{def}}{=} \{ \langle Q, Q' \rangle \mid Q, Q' \in \text{Q} \wedge \langle Q \rangle_+^\phi = \langle Q' \rangle_+^\phi \} \end{aligned}$$

So we can define the set of indistinguishable elements for HOANI:

$$\Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) = \{(\mathcal{J}(P', Q'))_+ \mid P' \equiv_{\eta} P \wedge Q' \equiv_{\phi} Q\}$$

These sets are collections of sets of traces that each secure abstraction must not distinguish, i.e. that must be approximated in the same object. Similarly we can define the predicate Secr for HOANI:

$$\begin{aligned} \forall X \in \wp(\Sigma^+) . \text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X) &\Leftrightarrow \forall P \in \mathsf{P} \forall Q \in \mathsf{Q} . \\ (\exists Z \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) . Z \subseteq X &\Rightarrow \forall W \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) . W \subseteq X) \end{aligned}$$

So $\text{Secr}(X)$ holds if X contains all the elements, i.e. all the sets of traces, that have to be indistinguishable or no one of this. Indeed Secr identifies all and only the sets which are in the secret kernel.

THEOREM 1.

Let $\eta, \phi \in \wp(\Sigma^+)$:

$$\mathcal{K}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(\text{id}) = \{X \in \wp(\Sigma^+) \mid \text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X)\}$$

PROOF. First, it is necessary to prove that this set is an upper closure operator and, after that, we have to prove that is the most concrete closure for which the program is safe. Upper closure operator are isomorphic to Moore family so we prove that $\{X \in \wp(\Sigma^+) \mid \text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X)\}$ is a Moore family. For doing so we have only to prove that thi set contains the intersection of all its elements. Consider $X, Y \in \{Z \in \wp(\Sigma^+) \mid \text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(Z)\}$. For hypothesis we have that:

$$\begin{aligned} \forall P \in \mathsf{P} \forall Q \in \mathsf{Q} . \\ (\exists Z \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) . Z \subseteq X &\Rightarrow \forall W \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) . W \subseteq X) \wedge \\ (\exists Z \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) . Z \subseteq Y &\Rightarrow \forall W \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q) . W \subseteq Y) \end{aligned}$$

So we have to prove that the same condition holds also for $X \cap Y$. Suppose that $\forall P \in \mathsf{P} \forall Q \in \mathsf{Q}$ it exists Z in $\Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q)$ such that $Z \in X \cap Y$ (indeed $Z \subseteq X$ and $Z \subseteq Y$). For hypothesis $\text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X)$ and $\text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(Y)$ hold, so $\forall W \in \Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P, Q)$ we have that W is contained in X and W is contained in Y , namely $W \subseteq X \cap Y$. Therefore $\text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X \cap Y)$ holds. This result can be easily extended to a generic intersection, so $\{X \in \wp(\Sigma^+) \mid \text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X)\}$ is Moore family. \checkmark

Now we have to prove that non-interference holds, i.e. for $\hat{\rho} \stackrel{\text{def}}{=} \{X \in \wp(\Sigma^+) \mid \text{Secr}_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(X)\}$ it is true $\text{H}_+[\eta]\mathcal{J}(\phi \Rightarrow \hat{\rho})_{\text{bca}}$. Suppose that it exists $P_1, P_2 \in \mathsf{P}$ and $Q_1, Q_2 \in \mathsf{Q}$ such that $\langle P_1 \rangle_+^{\eta} = \langle P_2 \rangle_+^{\eta} \wedge \langle Q_1 \rangle_+^{\phi} = \langle Q_2 \rangle_+^{\phi}$, with $\langle \mathcal{J}(P_1, Q_1) \rangle_+^{\hat{\rho}} \neq \langle \mathcal{J}(P_2, Q_2) \rangle_+^{\hat{\rho}}$. We can note that $\langle \mathcal{J}(P_1, Q_1) \rangle_+$ and $\langle \mathcal{J}(P_2, Q_2) \rangle_+$ are in $\Upsilon_{\mathcal{J},\eta,(\phi)}^{\text{H}^+}(P_1, Q_1)$, in fact $P_1 \equiv_{\eta} P_2$ and $Q_1 \equiv_{\phi} Q_2$. Let $X_1, X_2 \in \wp(\Sigma^+)$ defined as $X_1 \stackrel{\text{def}}{=} \langle \mathcal{J}(P_1, Q_1) \rangle_+^{\hat{\rho}}$ and $X_2 \stackrel{\text{def}}{=} \langle \mathcal{J}(P_2, Q_2) \rangle_+^{\hat{\rho}}$. For hypothesis we have that $X_1 \neq X_2$

but, due to $X_1, X_2 \in \hat{\rho}$, we also have that $\text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X_1)$ and $\text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X_2)$. Now, if $X_2 \not\subseteq X_1$, we have $(\mathcal{J}(P_1, Q_1))_+ \in \Upsilon_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(P_1, Q_1)$ such that $(\mathcal{J}(P_1, Q_1))_+ \subseteq X_1$, whilst $(\mathcal{J}(P_2, Q_2))_+ \in \Upsilon_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(P_1, Q_1)$ such that $(\mathcal{J}(P_1, Q_1))_+ \not\subseteq X_1$: absurd (for the hypothesis on X_1). Similarly, if $X_2 \subseteq X_1$ we obtain the same contradiction on X_2 . Therefore $(P_1)_+^\eta = (P_2)_+^\eta \wedge (Q_1)_+^\phi = (Q_2)_+^\phi$ implies $(\mathcal{J}(P_1, Q_1))_+^\hat{\rho} = (\mathcal{J}(P_2, Q_2))_+^\hat{\rho}$. \checkmark

Finally we have to prove that the closure is the most concrete. Suppose the contrary, i.e. it exists a domain ρ such that $\hat{\rho} \not\sqsubseteq \rho$ and ${}^{\text{H}}_+[\eta]\mathcal{J}(\phi \Rightarrow \rho)_{\text{bca}}$ holds. Remember that $\hat{\rho} \sqsubseteq \rho$ iff $\rho \subseteq \hat{\rho}$. Take in account the case in which $\hat{\rho} \subsetneq \rho$. At this point we can note that it exists $X \in \rho$ such that $X \notin \hat{\rho}$, i.e. for which $\text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X)$ doesn't hold. But this means that it exists $P \in \text{P}$ and $Q \in \text{Q}$ such that $\exists Z \in \Upsilon_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(P, Q) . Z \subseteq X$ and $\exists W \in \Upsilon_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(P, Q) . W \not\subseteq X$. So $Z = (\mathcal{J}(P_1, Q_1))_+$, for some $P_1 \in \text{P}, Q_1 \in \text{Q}$ and $W = (\mathcal{J}(P_2, Q_2))_+$, for some $P_2 \in \text{P}, Q_2 \in \text{Q}$, with $P_1 \equiv_\eta P_2 \equiv_\eta P$ and $Q_2 \equiv_\phi Q_2 \equiv_\phi Q$ due to the fact that both the set belong to $\Upsilon_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(P, Q)$. All this implies that $(\mathcal{J}(P_1, Q_1))_+^\rho \subseteq X$ due to $(\mathcal{J}(P_1, Q_1))_+ \in X$, whilst $(\mathcal{J}(P_2, Q_2))_+^\rho \not\subseteq X$ due to $(\mathcal{J}(P_2, Q_2))_+ \notin X$. Indeed $(\mathcal{J}(P_1, Q_1))_+^\rho \neq (\mathcal{J}(P_2, Q_2))_+^\rho$, but this is impossible, in fact we had supposed that ${}^{\text{H}}_+[\eta]\mathcal{J}(\phi \Rightarrow \rho)_{\text{bca}}$. So a domain like ρ doesn't exist. \checkmark \square

This means that the set of elements in $\wp(\Sigma^+)$ for which Secr holds it corresponds to the secret kernel of id , namely it coincides to the most concrete domain for which non-interference holds. This abstraction is called *most powerful harmless attacker for HOANI* and it is precisely the most precise attacker for which the program is safe. Furthermore we can characterize the secret kernel of a generic domain ρ .

COROLLARY 2.

Let $\eta, \phi \in \wp(\Sigma^+)$:

$$\mathcal{K}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(\rho) = \{X \in \rho \mid \text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X)\}$$

PROOF. By definition $\mathcal{K}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+} = \lambda\rho . \mathcal{K}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(\text{id}) \sqcup \rho$, so we have to prove that this domain is exactly $\{X \in \rho \mid \text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X)\}$. This means that $\{X \in \rho \mid \text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X)\}$ has to be equal to $\{X \in \wp(\Sigma^+) \mid \text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X)\} \sqcup \rho$. Remember that two sets are the same one is include in the other and vice versa. Consider an element $Y \in \{X \in \rho \mid \text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(X)\}$. Clearly for this element $Y \in \rho$ and $\text{Secr}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(Y)$ hold, i.e. Y belongs to $\mathcal{K}_{\mathcal{J}, \eta, (\phi)}^{\text{H}^+}(\text{id}) \sqcup \rho$. Similarly the inverse inclusion holds. \square

3 Formalization

Software watermarking consists in the embedding of some extra information, the *signature* or *watermark*, into a software for purposes such as intellectual property protection. In fact such watermark is hidden and so only

who is able to extract that information could claim the rights of the software. Moreover the signature should be indelible, in order to prevent the unwatermarking of the program. Looking at the software watermarking techniques present today we can group them in several ways: we can group them by purpose like authorship, fingerprinting, validation and licensing [21]; we can group them by satisfied properties like robustness/fragility, visibility/invisibility and multiplicity [21]; we can group them by extraction method like static and dynamic [2] (see [4, 2, 3, 21] for a taxonomy of watermarking techniques). Despite the classification, in every software watermarking technique can be identified three components. Let IMP be a programming language and \mathcal{S} a set of signatures. Following [8] we can define the nomenclature below¹.

1. **Stegomarker** $\mathfrak{M} : \mathcal{S} \rightarrow \text{IMP}$ – a function that generates a program which is the encoding of a given signature $s \in \mathcal{S}$, i.e. it generates the *stegomark* $\mathfrak{M}(s) \in \text{IMP}$.
2. **Stegoembedder** $\mathfrak{L} : \text{IMP} \times \text{IMP} \rightarrow \text{IMP}$ – a function that generates a program which is the composition of a stegomark and a cover program to sign, i.e. it generates the *stegoprogram* $\mathfrak{L}(P, \mathfrak{M}(s)) \in \text{IMP}$, with $P \in \text{IMP}$ and $s \in \mathcal{S}$.
3. **Stegoextractor** $\mathfrak{F} : \text{IMP} \rightarrow \wp(\mathcal{S})$ – a function that extracts the signatures from a stegoprogram; for all $s \in \mathcal{S}$ must be $s \in \mathfrak{F}(\mathfrak{L}(P, \mathfrak{M}(s)))$.

In the following, when \mathfrak{L} and \mathfrak{M} are clear from the context we denote the stegoprogram $\mathfrak{L}(P, \mathfrak{M}(s))$ as P_s .

In this work we focus the attention on the classification of watermarking techniques by insertion/extraction methods. This choice is led by the fact that if we want to discover when a stegoprogram can be unwatermarked or how a watermark can be avoided then we have to analyze how the secret information is hidden and which is the way for to reveal or to extract it. In *static watermarking* the signature is inserted in the cover program either as data (e.g. an image, a string, et cetera) or code (e.g. in the code control structure) so the stegomark can be extracted from the text of the program, without the need to execute it [2]. Conversely, in *dynamic watermarking* the signature is inserted in the program execution state (i.e. in its semantics) and so requires the program to be executed in order to extract the stegomark [2]. So in the latter case there is the concept of *enabling input*, namely a particular input sequence which makes the program enter a state which represents the watermark. In addition to this two categories, in [8] the authors proposed a technique which is different from both static and dynamic

¹The prefix “stego” stands for *steganographic*, in fact software watermarking is in general a steganographic method

watermarking. It is called *abstract watermarking* and the idea is that the stegomark is encoded in a way that it could be extracted by suitable static program analysis, that is by abstract interpretation. This technique can be seen like static watermarking because the extraction of the signature requires no execution. But, at the same time, it can be seen like dynamic watermarking because the stegomark is hidden in the semantics, though the execution of the stegoprogram will not reveal the signature. So abstract watermarking can be seen like both static and dynamic.

Starting from this observation, in [15] the author introduced the idea that static and dynamic watermarking are instances of abstract watermarking. In particular, they are instances of a common pattern which corresponds to the program transformations making semantics complete. Complete abstractions model precisely the full understanding of program semantics by an approximate observer, i.e. concrete computations can be replaced, with no loss of precision, with abstract one. So static and dynamic (and obviously abstract) watermarking techniques can be seen like program transformations which generates stegoprograms whose abstract semantics is complete. The completeness is referred to a specific program property which represents the stegomark. In this work we try to formalize and extend this idea in order to create a general framework for software watermarking, which is able to describe almost every possible watermarking technique.

3.1 The framework

As we have seen in section 2.3 we formalize the semantics of a program with its maximal finite traces semantics. This is reasonable because in software watermarking we are interested only in the terminating computations of a program and we need to keep track of *history* of such executions, for modeling the enabling input in dynamic watermarking. So the concrete domain of computation is $\wp(\Sigma^+)$ and the program properties are defined as upper closure operators in that domain. For to model dynamic techniques, we need to formalize the concept of enabling input. In fact these techniques reveal the signature only executing the stegoprogram with a particular input. We recall that IMP is a deterministic programming language and that the states of a trace embeds the residual input sequence left to be “consumed” by the program. So we can formalize the concept of enabling input like a state property, i.e. like an upper closure operator in the domain $\wp(\Sigma)$. Furthermore in static and abstract watermarking the extraction can be performed with static analysis and so with abstract interpretation. Our idea is that also in dynamic watermarking the stegoextractor can be implemented with abstract interpretation. Finally we assume that the set $\mathcal{P} \subseteq \text{IMP}$ of cover programs doesn’t contain programs that already encode a signature. Practically every software watermarking technique assumes to only deal with cover programs which are “free of signatures”, indeed, before the embed-

ding there is a preliminary step in which it is checked if the program to be signed is watermarkable.

So the stegoextractor takes a stegoprogram, analyses it and returns the signature encoded in the stegomark. This means that the encoded signature can be seen as a property of the stegomark's semantics and therefore of the stegoprogram's semantics. In this view a stegoextractor is an abstract interpreter that executes the stegoprogram in the abstract domain $\beta \in uco(\wp(\Sigma^*))$ that allows to observe the hidden signature. In order to deal with dynamic watermarking we need to model the enabling input that allows to extract the signature. Since the residual input stream is part of the program state, the enabling input can be modeled as a state property $\eta \in uco(\wp(\Sigma))$. We specify a watermarking system as a tuple $\langle \mathcal{L}, \mathfrak{M}, \beta \rangle$.

DEFINITION 3 (Software Watermarking System).

Given $\mathcal{L} : \text{IMP} \times \text{IMP} \rightarrow \text{IMP}$, $\mathfrak{M} : \mathcal{S} \rightarrow \text{IMP}$ and $\beta \in uco(\wp(\Sigma^+))$, the tuple $\langle \mathcal{L}, \mathfrak{M}, \beta \rangle$ is a software watermarking system for programs in \mathcal{P} and signatures in \mathcal{S} if \mathfrak{M} is injective and there exists $\eta \in uco(\wp(\Sigma))$ such that $\forall P \in \mathcal{P} \forall s \in \mathcal{S}$:

$$\begin{aligned} \llbracket \mathcal{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta &= \lambda X. \begin{cases} \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) & \text{if } X \in \eta(\wp(\Sigma^+)) \\ \llbracket P \rrbracket_+^\beta(X) & \text{otherwise} \end{cases} \\ X \in \eta(\wp(\Sigma^+)) &\Rightarrow \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta \end{aligned}$$

This means that when computing the semantics in the abstract domain β , the stegoprogram $\mathcal{L}(P, \mathfrak{M}(s))$ behaves like the stegomark $\mathfrak{M}(s)$ on the enabling inputs, and like the cover program P otherwise. It is possible to reduce the precise extraction of the signature to a completeness problem. To this end we associate the stegomarker $\mathfrak{M}(\cdot)$ with its semantic counterpart $\overline{\mathfrak{M}} : \mathcal{S} \rightarrow uco(\wp(\Sigma^+))$, which encodes a signature in a semantic program property. In particular, given watermarking system $\langle \mathcal{L}, \mathfrak{M}, \beta \rangle$ we define $\overline{\mathfrak{M}} \stackrel{\text{def}}{=} \lambda s. \{ \llbracket \mathfrak{M}(s) \rrbracket_+^\beta, \Sigma^+ \}$. Indeed, $\overline{\mathfrak{M}}(s)$ provides a semantic representation of the signature s . By construction we have that $\forall s \in \mathcal{S} : \beta \sqsubseteq \overline{\mathfrak{M}}(s)$ and this ensures that β is precise enough for extracting the signature. Moreover, the abstract semantics computed on β of the stegoprogram reveals the watermark $\overline{\mathfrak{M}}(s)$ under input η only if it is \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. This means that the stegoembedder makes programs in a way that the stegoextractor has a full comprehension of their semantics and so it is able to extract the property which represents the signature. Recalling the operator \mathbb{F} inducing \mathcal{F} -completeness introduced in section 2.2.1, we can say that if P_s is a stegoprogram then: $\llbracket P_s \rrbracket_+^\beta = \mathbb{F}_{\eta, \overline{\mathfrak{M}}(s)}(\llbracket P_s \rrbracket_+^\beta)$, i.e., $\llbracket P_s \rrbracket_+^\beta$ if \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. So, recalling the operator \mathbb{F} inducing \mathcal{F} -completeness introduced in section 2.2.1, we can say that if P_s is a stegoprogram then:

$$\llbracket P_s \rrbracket_+^\beta = \mathbb{F}_{\eta, \overline{\mathfrak{M}}(s)}(\llbracket P_s \rrbracket_+^\beta)$$

If \mathbb{P}_s^β is not \mathcal{F} -complete then $\mathbb{P}_s^\beta \circ \eta \neq \overline{\mathfrak{M}}(s) \circ \mathbb{P}_s^\beta \circ \eta$. Even if η is satisfied by X , we have that $\mathbb{P}_s^\beta(X) \neq \overline{\mathfrak{M}}(s) \circ \mathbb{P}_s^\beta(X)$ and so there is no doubt that $\mathbb{P}_s^\beta(X) \neq (\mathfrak{M}(s))^\beta$. Instead, if \mathbb{P}_s^β is \mathcal{F} -complete then $\mathbb{P}_s^\beta \circ \eta = \overline{\mathfrak{M}}(s) \circ \mathbb{P}_s^\beta \circ \eta$ holds. When η is satisfied by X , we have that $\mathbb{P}_s^\beta(x) = \overline{\mathfrak{M}}(s) \circ \mathbb{P}_s^\beta(x)$ and consequently we have that $\mathbb{P}_s^\beta(x) \in \overline{\mathfrak{M}}(s)$ (see Figure 1). This means that $\mathbb{P}_s^\beta(x)$ is an element of $\overline{\mathfrak{M}}(s)$ and so it represents the signature s . If η is not satisfied by X , the system should guarantee that the abstraction of the stegoprogram doesn't reveal the signature, so we have to choose β and in a way that $\mathbb{P}_s^\beta \notin \overline{\mathfrak{M}}(s)$, i.e. $\overline{\mathfrak{M}}(s)(\mathbb{P}_s^\beta(X)) = \Sigma^+$ minimizes false positive.

The different kinds of software watermarking techniques can be seen as instances of Definition 3.

Static and abstract software watermarking corresponds to $\eta = \text{id}$ and β decidable (i.e. implementable whit static analysis). In this case the interpretation of the stegoprogram always reveals the stegomark, independently from the input. Dynamic software watermarking corresponds to $\eta \neq \text{id}$ and β generic interpreter. In this case the concrete semantics of the stegoprogram reveals the stegomark only when a particular input sequence is given.

Note that if the abstract semantics of the stegoprogram is complete, it may well happen that the concrete semantics of the stegoprogram is not complete, i.e. \mathbb{P}_s^β is not \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. This means that the knowledge of the stegomarker may not be sufficient in order to extract the signature. Finally, due to the fact that the \mathbb{F} is idempotent, it provides also a tamper detection method. In fact $\mathbb{F}_{\eta, \overline{\mathfrak{M}}(s)}(\mathbb{P}_s^\beta) = \mathbb{P}_s^\beta$. Every syntactic transformation of the stegoprogram can be revealed. Let $t : \text{IMP} \rightarrow \text{IMP}$ be an attacker, if $\mathbb{F}_{\eta, \overline{\mathfrak{M}}(s)}(t(\mathbb{P}_s^\beta)) \neq \mathbb{P}_s^\beta$ then we can recognize that the stegoprogram is tampered.

3.2 Properties

For a software watermarking technique it is desirable to know if it is a good or bad technique or if it is better than another in some specific context of interest. So it is necessary to define some properties that could help when we want to measure the efficacy of a watermarking method and when we want to compare watermarking techniques. In this section we briefly describe and then we formalize the most significant properties that a software watermarking system could have, from a semantic point of view. Again we take advantage of abstract interpretation and so we complete the definition of the formal framework just introduced above. The properties that we will introduce are those which can be formalized semantically, not all the possible properties can be described in this way. For example, data-rate (capacity) [4] depends strictly by the implementation of the stegomarker, so it is not an intrinsic property of the watermarking system. Furthermore also

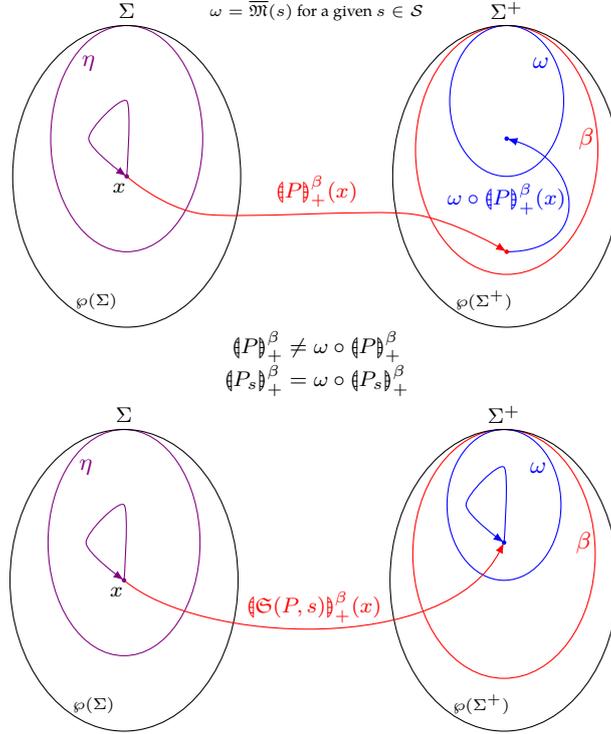


Figure 1: \mathcal{F} -completeness for the stegoprogram

credibility doesn't fit well in our semantic model, indeed it is a problem can be only handled in a statistical way.

In the follow, all the properties refer to a software watermarking system $\langle \mathcal{L}, \mathfrak{M}, \beta \rangle$, to a set of cover programs \mathcal{P} and to a set of signatures \mathcal{S} . With β we indicate the extraction domain and with $\overline{\mathfrak{M}}(s)$ we indicate the domain which encode semantically the signature s .

3.2.1 Resilience

Resilience concerns the capacity of a software watermarking system to be immune to a certain type of attacks. Mainly we can figure out four of such types.

Distortive attacks The attacker attempts to change the stegoprogram in order to compromise the stegomark, i.e. it applies syntactic or semantic transformations in order to make no more recoverable the signature by the stegoextractor.

Collusive attacks The attacker compares different stegoprograms of the same cover program in order to obtain informations on the stegomark.

Doing so it could, for example, identify the location of the stegomark within the stegoprogram and then remove it.

Subtractive attacks The attacker tries to eliminate the stegomark from the stegoprogram so that it is no longer possible to extract the signature (usually this attacks take place after that a preliminary analysis has identified the location of the stegomark).

Additive attacks The attacker adds another stegomark into the stegoprogram so that the previous stegomark is “overwritten” or so that the program contains more than one stegomark. In the latter case one can’t realize who has first marked the cover program and then the real author/owner can’t be determined.

Ideally, a software watermarking system is resilient, w.r.t. a particular kind of attack, when it is immune to any attack of that type. Furthermore we can divide the attacks in *conservative* and *not conservative*. The first maintain unmodified the semantics of the program under attack, whilst the second do not guarantee this (this attacks nullify the watermark but for doing so have to alter some functionalities of the program).

We have to do a preliminary clarification. Subtractive attacks need to locate the stegomark, so we fall back in problems of secrecy (explained below) and collusive attacks. Resilience to additive attacks is a non interesting case, indeed it is thought that software watermarking systems can’t be resilient to this type of attacks. Finally, collusive attacks are strictly related to the stegomark localization, so this type of resilience will be discussed with the concept of secrecy. So it remains only one type of attack then, from now, we consider the property of resilience as the resilience to distortive attacks.

An attacker can be seen like a program transformer $t : \text{IMP} \rightarrow \text{IMP}$ which modifies programs preserving their semantic, i.e. $\forall P \in \mathcal{P}$ must be that $\langle P \rangle_+ = \langle t(P) \rangle_+$. Truly the real objective of the attacker is to preserve only an abstraction of programs semantics, precisely the denotational semantics, i.e. to preserve the input/output programs behavior. In fact the attacker tries to modify as much as possible the program in order to compromise the stegomark. So there will be program’s properties that the attacker preserves and other that it does not preserve. So there will be abstractions $\psi \in \text{uco}(\wp(\Sigma^+))$ such that $\psi(\langle P \rangle_+) \neq \psi(\langle t(P) \rangle_+)$. Can be defined the most concrete property $\delta_t \in \text{uco}(\wp(\Sigma^+))$ preservable by t . So every property ψ more abstract, $\delta_t \sqsubseteq \psi$, will be clearly preserved by t .

If $\delta_t \sqsubseteq \bigcap \{ \overline{\mathfrak{M}}(s) \mid s \in \mathcal{S} \}$ then every stegoprogram, so the software watermarking system, is protected against the attacker t . Otherwise, could be that t preserves $\overline{\mathfrak{M}}(s)$ for certain signatures, in particular for those which $\delta_t \sqsubseteq \overline{\mathfrak{M}}(s)$. So we can characterize which stegoprograms are immune to t and which are not. In the worst case, when $\forall s \in \mathcal{S} . \delta_t \not\sqsubseteq \overline{\mathfrak{M}}(s)$, the software watermarking system is not able to contrast in any way the attacker t .

If, for some s , $\delta_t \not\sqsubseteq \overline{\mathfrak{M}}(s)$ we can measure how much the stegoprogram is vulnerable w.r.t. t . We can calculate how much information about $\overline{\mathfrak{M}}(s)$ is modifiable: $\widetilde{\mathfrak{M}}(s) = \overline{\mathfrak{M}}(s) \ominus (\delta_t \sqcup \overline{\mathfrak{M}}(s))$ (see [11] for the details). Most concrete is $\widetilde{\mathfrak{M}}(s)$ and less resilient the stegoprogram is. Similarly we can work on $\prod\{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$, measuring how much the whole system is vulnerable. So, to summarize, the property $\prod\{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$ induces a partition of the possible attackers, in the following way.

DEFINITION 4 (t-resilience).

A software watermarking system is:

- ▷ t-resilient if $\delta_t \sqsubseteq \prod\{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$
- ▷ t-vulnerable if $\exists s \in \mathcal{S} . \delta_t \not\sqsubseteq \overline{\mathfrak{M}}(s)$
- ▷ t-ineffective if $\forall s \in \mathcal{S} . \delta_t \not\sqsubseteq \overline{\mathfrak{M}}(s)$

Furthermore, if an attacker t is conservative it must preserve the denotational semantics, $\text{DenSem} \in \text{uco}(\wp(\Sigma^+))$ ², of the original program so we have that $\delta_t \sqsubseteq \text{DenSem}$. In this context every property more abstract than DenSem is preserved.

DEFINITION 5 (Resilience).

A software watermarking system is resilient if

$$\text{DenSem} \sqsubseteq \prod\{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$$

Therefore, in this last case, the system is t-resilient w.r.t any distortive (and conservative) attacker t . A software watermarking system which exhibits such behavior has not been yet found and it is an open research topic to demonstrate its existence or not.

If the attacker is not conservative, so it is willing to lose some original program functionality in order to nullify the stegomark, then is not necessary that $\delta_t \sqsubseteq \text{DenSem}$. In this case we have a stronger formalization of the attacker and it is not possible to assert that a software watermarking system is resilient to any distortive attacker not conservative.

This formalization of resilience allows us to compare two watermarking systems w.r.t. resilience. Given two software watermarking systems $\mathfrak{A}_1 = \langle \mathcal{L}_1, \mathfrak{M}_1, \beta_1 \rangle$ and $\mathfrak{A}_2 = \langle \mathcal{L}_2, \mathfrak{M}_2, \beta_2 \rangle$, if it holds that $\prod\{\overline{\mathfrak{M}}_1(s) \mid s \in \mathcal{S}\} \sqsubseteq \prod\{\overline{\mathfrak{M}}_2(s) \mid s \in \mathcal{S}\}$ then we have that $\{t \mid \delta_t \sqsubseteq \{\overline{\mathfrak{M}}_1(s) \mid s \in \mathcal{S}\}\}$ is contained in $\{t \mid \delta_t \sqsubseteq \{\overline{\mathfrak{M}}_2(s) \mid s \in \mathcal{S}\}\}$. Therefore \mathfrak{A}_2 is, in general, more resilient than \mathfrak{A}_1 .

²This domain is obtained from maximal finite traces semantics by the abstraction $\text{DenSem}_\alpha(x) \stackrel{\text{def}}{=} \{\sigma \in \Sigma^+ \mid \exists \sigma' \in x . \sigma_{\vdash} = \sigma'_{\vdash} \wedge \sigma_{\dashv} = \sigma'_{\dashv}\}$

3.2.2 Secrecy

Secrecy concerns how much is hard to recover the stegomark embedded in a stegoprogram. Ideally a software watermarking system is secret when is impossible to extract the signature (and so the stegomark) from a stegoprogram without knowing the stegoextractor. In practice, secrecy can be seen as the ability of the software watermarking system to make indistinguishable a set of signatures to the attacker. In this way the attacker is also not able to extract such signatures. Here comes the resilience to collusive attacks. A watermarking system is resilient to collusive attacks when an attacker is not able to view the difference between different stegoprograms related to the same cover program (these stegoprograms contain different stegomark and so different signatures). This concept can be formalized with *higher-order abstract non-interference*, introduced in section 2.4.

The private input is the set of programs that have to be secret, i.e. all possible stegomark, so $Q = \{\mathfrak{M}(s) \mid s \in \mathcal{S}\}$. Instead the public input is the set of cover programs, so $P = \mathcal{P}$. Now we can consider $\phi \in uco(\wp(\Sigma^+))$, a property that represents some stegomark, and so a set of signatures. We assume that the attacker doesn't have access to cover programs, so the abstraction of the public input is id . The system is safe if it holds non-interference ${}^H_+[\text{id}]\mathfrak{L}(\phi \Rightarrow \rho)_{\text{bca}}$.

DEFINITION 6 (ϕ -secrecy).

A software watermarking system is ϕ -secret, w.r.t. attacker ρ , if ${}^H_+[\text{id}]\mathfrak{L}(\phi \Rightarrow \rho)_{\text{bca}}$ holds, i.e. if

$$\forall P \in \mathcal{P}, \forall Q_1, Q_2 \in \mathcal{Q}. \\ \llbracket Q_1 \rrbracket_+^\phi = \llbracket Q_2 \rrbracket_+^\phi \Rightarrow \llbracket \mathfrak{L}(P, Q_1) \rrbracket_+^\rho = \llbracket \mathfrak{L}(P, Q_2) \rrbracket_+^\rho$$

This means that if we codify a cover program with two different signatures, which have the same property ϕ , then an attacker is not able to view differences between the stegoprograms generated, assuming that its observation power is ρ . In this way all the signatures with the same property ϕ can be used for generating stegoprograms resilient to collusive attacks from the attacker ρ . The system is more secure, w.r.t. the attacker ρ , the greater is the set of signatures for which it holds the non-interference defined above. In particular when $\phi = \top$, with $\top = \lambda x. \Sigma^+$, the system is said to be *secret* w.r.t. the observer ρ . In this case the set of indistinguishable signatures is equal to \mathcal{S} and no properties can flow.

DEFINITION 7 (Secrecy).

A software watermarking system is secret, w.r.t. attacker ρ , if ${}^H_+[\text{id}]\mathfrak{L}(\top \Rightarrow \rho)_{\text{bca}}$ holds, i.e. if

$$\forall P \in \mathcal{P}, \forall Q_1, Q_2 \in \mathcal{Q}. \llbracket \mathfrak{L}(P, Q_1) \rrbracket_+^\rho = \llbracket \mathfrak{L}(P, Q_2) \rrbracket_+^\rho$$

Moreover, if we fix the property ϕ , i.e. a set of signatures, we can analyze for which attacker the system is ϕ -secret. In this way we can characterize the most concrete observer $\hat{\rho}$ for which the non-interference $\mathcal{L}(\phi \Rightarrow \hat{\rho})_{\text{bca}}^{\text{H}+[id]}$ holds, called *most powerful ϕ -secret attacker*³. Clearly the analysis is useful for any observer most concrete, or concrete as, β . In fact the system can't be secret for the attackers at least precise as the extractor.

Thus, the secrecy level of a watermarking system is given by the most abstract property ϕ and by the most concrete observer $\hat{\rho}$ for which non-interference $\mathcal{L}(\phi \Rightarrow \hat{\rho})_{\text{bca}}^{\text{H}+[id]}$ holds. The more ϕ is abstract and the more the system is secret. Vice versa, the more $\hat{\rho}$ is concrete and the more the system is secret. Observe that ϕ can range from id (all the signatures are distinguishable) to \top (no signature is distinguishable). If $\hat{\rho}$, i.e., the most powerful ϕ -secret attacker, is \top then every attacker is able to distinguish the signatures. Otherwise, the more $\hat{\rho}$ is less abstract w.r.t. \top the more the system is secret.

This formalization of secrecy allows us to compare two watermarking systems w.r.t. secrecy. Given two software watermarking systems $\mathfrak{A}_1 = \langle \mathcal{L}_1, \mathfrak{M}_1, \beta_1 \rangle$ and $\mathfrak{A}_2 = \langle \mathcal{L}_2, \mathfrak{M}_2, \beta_2 \rangle$, and consider the most powerful harmless attacker. If $\hat{\rho}_1 \sqsubseteq \hat{\rho}_2$ we have that \mathfrak{A}_1 is more secret, w.r.t. ϕ , than \mathfrak{A}_2 . Indeed it is necessary a more precise (so most stronger) attacker for to void ϕ -secrecy with \mathfrak{A}_1 .

3.2.3 Transparency

Transparency concerns the ability to make hard to discover if a generic program is a stegoprogram. Ideally, a software watermarking system is transparent when it is impossible to view differences between a program and the related stegoprogram. So we can say that a software watermarking system is invisible, w.r.t. a fixed observer, if the latter is not able to distinguish a generic cover program from every stegoprogram generated starting from it.

DEFINITION 8 (Invisibility).

A software watermarking system is invisible, w.r.t. attacker ρ , if

$$\forall P \in \mathcal{P} \forall s \in \mathcal{S}. \langle P \rangle_+^\rho = \langle P_s \rangle_+^\rho$$

The greatest is the set of the observers for which the system is invisible and the greatest is the level of transparency. So the characterization of the most concrete observer $\hat{\rho}$ for which the system is invisible is a good measure of the transparency of the software watermarking system. This observer $\tilde{\rho}$ is called *most powerful transparent attacker*. This attacker can be characterized with a slightly differentiation of the most powerful \top -secret attacker. In

³This attacker can be derived calculating the secret kernel of higher-order abstract non-interference for traces semantics, like showed in section 2.4

fact a system, in order to be invisible w.r.t. an attacker has clearly to be also \top -secret w.r.t. that attacker. So, recalling the set of indistinguishable elements for HOANI $\Upsilon_{\mathcal{L},\eta,(\phi)}^{\text{H}+}(P, Q)$ introduced in Section 2.4, we can define its counterpart for transparence, namely all the elements which have to be indistinguishable for transparence. This is

$$\Upsilon_{\mathcal{L},id,(\top)}^{\text{H}+}(P, \mathfrak{M}(s)) = \{(\mathcal{L}(P', \mathfrak{M}(s'))\}_+ | P' \equiv_{id} P \wedge \mathfrak{M}(s') \equiv_{\top} \mathfrak{M}(s)\} \cup \{(P)\}_+$$

These sets are collections of sets of traces that each secure abstraction must not distinguish, i.e. that must be approximated in the same object. Then we can continue the construction of the secret kernel as done in Section 2.4. Clearly the analysis is useful for any observer most concrete, or concrete as, β . In fact the system can't be invisible for the attackers at least precise as the extractor.

Similarly to what we have done for secrecy, given two software watermarking systems $\mathfrak{A}_1 = \langle \mathcal{L}_1, \mathfrak{M}_1, \beta_1 \rangle$ and $\mathfrak{A}_2 = \langle \mathcal{L}_2, \mathfrak{M}_2, \beta_2 \rangle$, and their most powerful harmless attackers we have that if $\mathcal{K}_{\mathcal{L}_1}(id) \sqsubseteq \mathcal{K}_{\mathcal{L}_2}(id)$ we have that \mathfrak{A}_1 is more transparent than \mathfrak{A}_2 .

3.2.4 Accuracy

Informally, a software watermarking system is accurate if preserves the functionalities of the cover program, i.e. the cover program and the stegoprogram have to exhibit the same *observable behavior*. This concept can be defined as "behavior as experienced by the user" [4]. Precisely, the stegoprogram can do something (file modifications, additional network traffic, et cetera) that the cover program doesn't do, but this *side-effects* must be not visible to the user. Clearly this definition is very loose and it depends on what the user is able to observe regard the program execution.

A possible characterization, at semantic level, of this concept can be the follow: the stegoprogram and the original program must have the same *observable* denotational semantics. This means that, fixed what the user is interested to (or is able to) observe, the stegoprogram and the cover program must exhibit the same input/output behavior w.r.t. the fixed observation level.

Formally, we define an observational abstraction $\alpha_{\mathcal{O}}$ which characterizes what is interesting to observe about the denotational semantics of programs. Then it is required, so worth the accuracy, that the cover program P and a generic stegoprogram P_s are equivalent modulo this abstraction, i.e. $\alpha_{\mathcal{O}}(\llbracket P \rrbracket_{Den}) = \alpha_{\mathcal{O}}(\llbracket P_s \rrbracket_{Den})$ [9].

Then let $\langle D_{\mathcal{O}}, \leq_{\mathcal{O}} \rangle$ be a poset and $\alpha_{\mathcal{O}} : \wp(\Sigma) \rightarrow D_{\mathcal{O}}$ be a function such that $(\langle \wp(\Sigma), \subseteq \rangle, \alpha_{\mathcal{O}}, \alpha_{\mathcal{O}}^+, \langle D_{\mathcal{O}}, \leq_{\mathcal{O}} \rangle)$ is a Galois connection. We say that two programs $P, Q \in \text{IMP}$ are $\alpha_{\mathcal{O}}$ -*observationally equivalent*, regard the denotational semantics, if and only if $\alpha_{\mathcal{O}}(\llbracket P \rrbracket_{Den}) = \alpha_{\mathcal{O}}(\llbracket Q \rrbracket_{Den})$.

Returning to software watermarking, a system is accurate, fixed an observational abstraction $\alpha_{\mathcal{O}}$, if for each program $P \in \mathcal{P}$ and for each signature $s \in \mathcal{S}$ is true that P is $\alpha_{\mathcal{O}}$ -observationally equivalent to P_s .

DEFINITION 9 (Accuracy).

Given a poset $\langle D_{\mathcal{O}}, \leq_{\mathcal{O}} \rangle$ and an observational abstraction $\alpha_{\mathcal{O}} : \wp(\Sigma) \rightarrow D_{\mathcal{O}}$, such that $\langle \wp(\Sigma), \subseteq \rangle \xrightleftharpoons[\alpha_{\mathcal{O}}]{\alpha_{\mathcal{O}}^{\dagger}} \langle D_{\mathcal{O}}, \leq_{\mathcal{O}} \rangle$, we have that a software watermarking system is accurate, w.r.t. $\alpha_{\mathcal{O}}$, if:

$$\forall P \in \mathcal{P} \forall s \in \mathcal{S} . \alpha_{\mathcal{O}}(\llbracket P \rrbracket_{Den}) = \alpha_{\mathcal{O}}(\llbracket P_s \rrbracket_{Den})$$

For example, a reasonable observational abstraction could be the output given to the user. In the language IMP there is a command for to catch the values inserted by the user, but there is not a command for to show the results to the user. This can be simulated restricting a subset of a program's variables to store the values that have to be showed to the user. We can consider $\text{Var}_{out}(P) \subseteq \text{var} \llbracket P \rrbracket$ the set of this kind of variables. The user who wants to observe the output of the program P has easily to check the values of the variables in this set. IMP provides functions $\rho \in \text{Env}$ (environments) which represent, in a determinate state, the binding from variables and values⁴. So, in a determinate state $\langle C, \langle \rho, \iota \rangle \rangle$ of program P , $\text{var} \llbracket P \rrbracket = \text{dom}(\rho)$. We define $\text{dom}_{out}(\rho) \subseteq \text{dom}(\rho)$ the set of output variables of environment ρ . The abstraction that catches the output given to the user observes only the values of those variables, so $\alpha_{\mathcal{O}} \stackrel{\text{def}}{=} \alpha_{\mathcal{O}}^{out} : \wp(\Sigma) \rightarrow \wp(\text{Env})$ can be defined in the following way:

$$\alpha_{\mathcal{O}}^{out}(X) \stackrel{\text{def}}{=} \left\{ \rho' \in \text{Env} \mid \exists \varsigma \in X . \begin{array}{l} \varsigma = \langle C, \langle \rho, \iota \rangle \rangle, \\ \text{dom}(\rho') = \text{dom}_{out}(\rho), \\ \forall y \in \text{dom}_{out}(\rho) . \rho'(y) = \rho(y) \end{array} \right\} \quad (1)$$

In this case, $\alpha_{\mathcal{O}}^{out}(\llbracket P \rrbracket_{Den})$ catches the input/output behavior, specified as input inserted by the user and output showed to the user, of program P ⁵. If we apply $\alpha_{\mathcal{O}}^{out}$ to definition 9 then we obtain that a software watermarking system is accurate, w.r.t. $\alpha_{\mathcal{O}}^{out}$, if:

$$\forall P \in \mathcal{P} \forall s \in \mathcal{S} . \alpha_{\mathcal{O}}^{out}(\llbracket P \rrbracket_{Den}) = \alpha_{\mathcal{O}}^{out}(\llbracket P_s \rrbracket_{Den})$$

As regarding accuracy, this is a property that is not directly comparable among different watermarking techniques since it is defined w.r.t. the observational abstraction of interest.

⁴See section 2.3 for the details

⁵Clearly this observation is useless for the programs which don't interact with the user; in this cases it is necessary to chose another type of observable abstraction, more suitable for the context

4 Validation

In this section we describe five common watermarking techniques and we show how they can be formalized with our framework. These techniques are voluntarily chosen heterogeneous, in order to represent all software watermarking typology (static, dynamic and abstract). In the follow, we indicate with $\prod \overline{\mathfrak{M}}$ the reduced product of all the abstract stegomark $\overline{\mathfrak{M}}(s)$, i.e. $\prod \overline{\mathfrak{M}} \stackrel{\text{def}}{=} \{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$.

4.1 Static graph-based watermarking

Static software watermarking technique, conceived by *Venkatesan et al.* [22], where the signature (a natural number) is codified as a graph which is added to the CFG⁶ of the cover program. In particular, a program whose CFG is equal to the graph generated starting from the signature is derived and then added to cover program in a way that its semantics remains unmodified, like it is showed in figure 2. The nodes of the added graph are marked before the embedding, in order to be distinguishable from the nodes of the original CFG at the extraction step.

Let $\mathcal{E} : \mathbb{N} \rightarrow \mathbb{G}$ be a function that codify a signature in a graph. Let $\text{Mark} : \Sigma^+ \rightarrow \mathbb{G}$ be a function that, given a trace σ , outputs the marked subgraph of the CFG of σ for a certain marking criterion⁷. The semantics $(P)_+^\beta$ extracts the marked subgraph of the CFG of P , so the extraction domain β is:

$$\beta \stackrel{\text{def}}{=} \{X \in \wp(\Sigma^+) \mid \exists g \in \mathbb{G}. X = \{\sigma \in \Sigma^+ \mid \text{Mark}(\sigma) = g\}\} \cup \{\Sigma^+\}$$

So in β there are all the sets of traces whose CFG contains the same marked graph. With $\mathcal{W}_s \stackrel{\text{def}}{=} \{\sigma \in \Sigma^+ \mid \mathcal{E}(s) = \text{Mark}(\sigma)\}$ we indicate the set of traces whose CFG contains the marked graph which codify the signature s . This is a static technique so $\eta = \text{id}$. Clearly $(\overline{\mathfrak{M}}(s))_+^\beta = \mathcal{W}_s$ and so $\overline{\mathfrak{M}}(s) = \{\mathcal{W}_s, \Sigma^+\}$. Let $\overline{\mathbb{G}} \stackrel{\text{def}}{=} \{\perp, \mathbb{G}\} \cup \{\mathbb{G}\}$. The domain β can be defined as $\beta \stackrel{\text{def}}{=} \beta_\gamma \circ \beta_\alpha$ where $\beta_\alpha : \wp(\Sigma^+) \rightarrow \overline{\mathbb{G}}$ and $\beta_\gamma : \overline{\mathbb{G}} \rightarrow \wp(\Sigma^+)$ are

$$\beta_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} \perp & \text{if } X = \emptyset \\ g & \text{if } \forall \sigma \in X. g = \text{Mark}(\sigma) \\ \mathbb{G} & \text{otherwise} \end{cases}$$

$$\beta_\gamma \stackrel{\text{def}}{=} \lambda g. \begin{cases} \emptyset & \text{if } g = \perp \\ \{\sigma \in \Sigma^+ \mid g = \text{Mark}(\sigma)\} & \text{if } g \in \mathbb{G} \\ \Sigma^+ & \text{otherwise} \end{cases}$$

⁶Control Flow Graph

⁷Building the CFG and locating its marked nodes are both tasks easily implementable analyzing a program traces

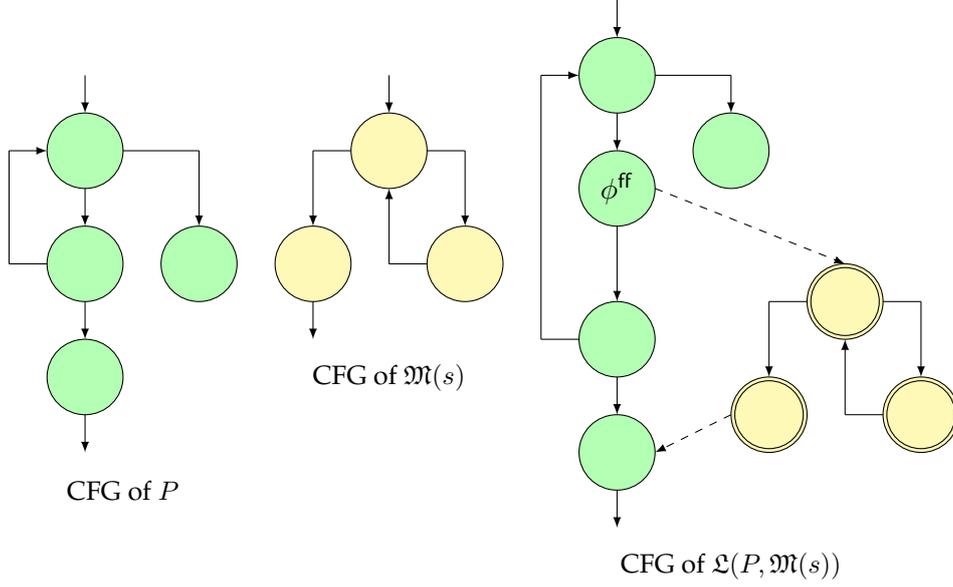


Figure 2: Static graph-based watermarking

Instead $\overline{\mathfrak{M}}(s)$ can be defined as $\overline{\mathfrak{M}}(s) \stackrel{\text{def}}{=} \overline{\mathfrak{M}}(s)_\gamma \circ \overline{\mathfrak{M}}(s)_\alpha$ where $\overline{\mathfrak{M}}(s)_\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ and $\overline{\mathfrak{M}}(s)_\gamma : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ are

$$\overline{\mathfrak{M}}(s)_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} \emptyset & \text{if } X = \emptyset \\ \mathcal{W}_s & \text{if } X \subseteq \mathcal{W}_s \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \overline{\mathfrak{M}}(s)_\gamma \stackrel{\text{def}}{=} \lambda X. X$$

It is simple to see that for all signature s we have $\overline{\mathfrak{M}}(s)(\wp(\Sigma^+)) \subseteq \beta(\wp(\Sigma^+))$ and so $\beta \sqsubseteq \overline{\mathfrak{M}}(s)$. Then we have to check if \mathfrak{L} is a stegoembedder w.r.t. β . The input domain is id so we have to check if for all set of states X we have

$$\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) \wedge \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X)$$

Clearly for every possible set of initial states, the CFG of $\mathfrak{L}(P, \mathfrak{M}(s))$ is the same, i.e. it exists $g \in \mathbb{G}$ such that $\forall \sigma \in \llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+$ we have $g = \text{CFG}(\sigma)$. For how the technique is designed, into g there is a marked subgraph equal to $\mathcal{E}(s)$. So we have that $\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states. Now, the CFG of $\mathfrak{M}(s)$ is exactly $\mathcal{E}(s)$ and it is marked by design, so $\llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states. Therefore we have $\forall X \in \wp(\Sigma)$.

$$\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X)$$

We can also note that, for every signature s , $\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta$ is \mathcal{F} -complete

for η and $\overline{\mathfrak{M}}(s)$. In fact we have:

$$\begin{aligned} \mathfrak{P}_+^\beta(X) \circ \eta &= \overline{\mathfrak{M}}(s) \circ \mathfrak{P}_+^\beta(X) \circ \eta \\ \mathcal{W}_s &= \overline{\mathfrak{M}}(s)(\mathcal{W}_s) \\ \mathcal{W}_s &= \mathcal{W}_s \end{aligned}$$

Now let's see the properties of this technique. The system is not resilient, because it is not fully immune to distortive attacks, i.e. $\text{DenSem} \not\sqsubseteq \prod \{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$.

PROOF. Suppose that $\text{DenSem} \sqsubseteq \prod \overline{\mathfrak{M}}$, so $\forall X \in \wp(\Sigma^+)$ it holds that $\text{DenSem}(X) \subseteq \prod \overline{\mathfrak{M}}(X)$. Let $X = \mathcal{W}_s$, for a generic signature s , so $\prod \overline{\mathfrak{M}}(X) = \mathcal{W}_s$. But $\mathcal{W}_s \subsetneq \text{DenSem}(\mathcal{W}_s)$, because there is for sure at least a trace with the same initial and final state of a trace in \mathcal{W}_s but with $\text{Mark}(\sigma) \neq \mathcal{E}(s)$. For example, take a program equals to $\mathcal{L}(P, \mathfrak{M}(s))$ but in which all the nodes of its CFG are unmarked. Clearly its traces are in $\text{DenSem}(\mathcal{W}_s)$ but they aren't in \mathcal{W}_s , because this traces don't have a marked subgraph. So there is a X for which $\text{DenSem}(X) \not\subseteq \prod \overline{\mathfrak{M}}(X)$ and hence $\text{DenSem} \not\sqsubseteq \prod \overline{\mathfrak{M}}$. \square

Indeed the system is vulnerable to control flow obfuscation techniques (basically the ones which modify the CFG). For example, a CFG flattening attack is able to damage the stegomark. Let's try to derive the most powerful \top -secret attacker for this technique. Like showed in Section 2.4, the most powerful \top -secret attacker for static graph-based watermarking is

$$\mathcal{K}_{\mathcal{L},(\text{id}),\top}(\text{id}) = \{X \in \wp(\Sigma^+) \mid P \in \mathcal{P}, X = \bigcup_{Q \in \mathcal{Q}} (\mathcal{L}(P, Q))_+ \} \cup \{\Sigma^+\}$$

and it abstracts in the same object the traces of all possible stegoprograms related to the same cover program. Finally the system is also $\alpha_{\mathcal{O}}^{\text{out}}$ accurate, indeed the behavior of the cover program is preserved w.r.t. this observation. Clearly for every program P and for every signature s it holds that $\alpha_{\mathcal{O}}^{\text{out}}((P)_{\text{Den}}) = \alpha_{\mathcal{O}}^{\text{out}}((P_s)_{\text{Den}})$. In fact the insertion of the marked graph doesn't affect the variables of the cover program (the added code is never executed). So, with the same input, the cover program and the stegoprogram give the same output.

4.2 Path-based watermarking

Dynamic software watermarking technique, conceived by *Collberg et al.* [5], where the signature (a natural number) is encoded by the sequence of choices made at conditional statements during a particular execution of the program. This execution is generated by a particular sequence of input values I_0, I_1, \dots, I_k called enabling sequence. The embedder takes the program code and it adds bogus branches in a way that the sequence of choices at conditional statements make by the resulting program, with the enabling,

input, is equal to the binary notation of the signature. With ϵ we indicate the empty sequence. Let $\text{Bin} : \mathbb{N} \rightarrow \{0, 1\}^*$ be a function that returns the binary encoding of a natural number and $\text{Branch} : \Sigma^+ \rightarrow \{0, 1\}^*$ be a function that extracts the sequence of choices at conditional statements in a trace. For example, when the guard of an instruction is evaluated to tt it can be assigned to this choice the value 1 and it can be assigned the value 0 if the guard is evaluated to ff. Let $\mathcal{E} : \mathbb{N} \rightarrow \wp(\Sigma^+)$ ⁸ the function:

$$\mathcal{E} \stackrel{\text{def}}{=} \lambda k. \left\{ \sigma \in \Sigma^+ \mid \begin{array}{l} |\sigma| = n + 1, \text{Branch}(\sigma) = \text{Bin}(k), \\ \sigma_n = \langle C, \langle \rho_n, \iota_n \rangle \rangle, \text{top}(\iota_i) = \epsilon \end{array} \right\}$$

The semantics $(P)_+^\beta$ extracts the sequence of choices at conditional statements for the program P , so the domain β is

$$\beta \stackrel{\text{def}}{=} \{X \in \wp(\Sigma^+) \mid k \in \{0, 1\}^*, X = \mathcal{E}(k)\} \cup \{\Sigma^+\}$$

and it contains all the sets of traces which have done the same choices, when all the input values are consumed. With $\mathcal{W}_s = \mathcal{E}(s)$ we indicate the set of traces for which, when all the input values are consumed, the sequence of choices at conditional statements codify the signature s . This is a dynamic technique, so $\eta = \wp(\mathcal{I}) \cup \{\Sigma^+\}$, where \mathcal{I} represents the set of states enabling the watermark, i.e.

$$\mathcal{I} \stackrel{\text{def}}{=} \left\{ \varsigma \in \Sigma \mid \begin{array}{l} \varsigma = \langle C, \langle \rho, \iota \rangle \rangle, |\iota| = |I|, \\ \forall j \in [0, |I|]. \text{top}(\text{next}(\iota)^j) = I_j \end{array} \right\}$$

The domain η can be defined as $\eta \stackrel{\text{def}}{=} \eta_\gamma \circ \eta_\alpha$ where $\eta_\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ and $\eta_\gamma : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ are

$$\eta_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} X & \text{if } X \subseteq \mathcal{I} \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \eta_\gamma \stackrel{\text{def}}{=} \lambda X. X$$

Clearly $(\overline{\mathfrak{M}}(s))_+^\beta = \mathcal{W}_s$ and so $\overline{\mathfrak{M}}(s) = \{\mathcal{W}_s, \Sigma^+\}$. Let $\overline{\mathbb{N}} \stackrel{\text{def}}{=} \{\perp, \mathbb{N}\} \cup \{\mathbb{N}\}$. The domain β can be defined as $\beta \stackrel{\text{def}}{=} \beta_\gamma \circ \beta_\alpha$ where $\beta_\alpha : \wp(\Sigma^+) \rightarrow \overline{\mathbb{N}}$ e $\beta_\gamma : \overline{\mathbb{N}} \rightarrow \wp(\Sigma^+)$ are

$$\beta_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} \perp & \text{if } X = \emptyset \\ k \in \mathbb{N} & \text{if } \forall \sigma \in X. \begin{array}{l} |\sigma| = n + 1, \sigma_n = \langle C_n, \langle \rho_n, \iota_n \rangle \rangle \\ \text{top}(\iota_n) = \epsilon, \text{Branch}(\sigma) = \text{Bin}(k) \end{array} \\ \mathbb{N} & \text{otherwise} \end{cases}$$

$$\beta_\gamma \stackrel{\text{def}}{=} \lambda k. \begin{cases} \emptyset & \text{if } k = \perp \\ \mathcal{E}(k) & \text{if } k \in \mathbb{N} \\ \Sigma^+ & \text{otherwise} \end{cases}$$

⁸This function is not necessary, it is only useful for a simpler reading of the text

Instead $\overline{\mathfrak{M}}(s)$ can be defined as $\overline{\mathfrak{M}}(s) \stackrel{\text{def}}{=} \overline{\mathfrak{M}}(s)_\gamma \circ \overline{\mathfrak{M}}(s)_\alpha$ where $\overline{\mathfrak{M}}(s)_\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ and $\overline{\mathfrak{M}}(s)_\gamma : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ are

$$\overline{\mathfrak{M}}(s)_\alpha \stackrel{\text{def}}{=} \lambda S. \begin{cases} \mathcal{W}_s & \text{if } X \subseteq \mathcal{W}_s \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \overline{\mathfrak{M}}(s)_\gamma \stackrel{\text{def}}{=} \lambda X. X$$

It is simple to see that for all signature s we have $\overline{\mathfrak{M}}(s)(\wp(\Sigma^+)) \subseteq \beta(\wp(\Sigma^+))$ and so $\beta \sqsubseteq \overline{\mathfrak{M}}(s)$. Then we have to check if \mathfrak{L} is a stegoembedder w.r.t. β . The input domain is not id so we have to check if for all set of states X we have

$$\begin{aligned} \spadesuit X \in \eta(\Sigma) &\Rightarrow \llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) \wedge \llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \llbracket \mathfrak{M}(s) \rrbracket_+^\beta \\ \clubsuit X \notin \eta(\Sigma) &\Rightarrow \llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \llbracket P \rrbracket_+^\beta \end{aligned}$$

If $X \in \eta(\Sigma)$ then $X \subseteq \mathcal{I}$. All such X contain states which encode the enabling input, so the choices at conditional statements made by $\mathfrak{L}(P, \mathfrak{M}(s))$ starting from states in X are equal to $\text{Bin}(s)$. So we have that $\llbracket \mathfrak{L}(L, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states that satisfy η . Now, the same reasoning can be done for $\mathfrak{M}(s)$, because it codify the signature by design (starting from the sets of input states which encode the enabling input). So $\llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \mathcal{W}_s$ for every $X \in \eta(\Sigma)$. Therefore we have that \spadesuit holds. If $X \notin \eta(\Sigma)$ then $X \not\subseteq \mathcal{I}$. All such X don't contain states which encode the enabling input, so the choices at conditional statements made by $\mathfrak{L}(P, \mathfrak{M}(s))$ starting from states in X are not equal to $\text{Bin}(s)$. This choices are with high probability the same of those made by P . Even better, it is very likely that both $\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X)$ and $\llbracket P \rrbracket_+^\beta$ are equal to Σ^+ . Indeed we have that \clubsuit holds.

We can also note that, for every signature s , $\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta$ is \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. In fact we have, when $X \in \eta(\Sigma)$:

$$\begin{aligned} \llbracket P \rrbracket_+^\beta(X) \circ \eta &= \overline{\mathfrak{M}}(s) \circ \llbracket P \rrbracket_+^\beta(X) \circ \eta \\ \mathcal{W}_s &= \overline{\mathfrak{M}}(s)(\mathcal{W}_s) \\ \mathcal{W}_s &= \mathcal{W}_s \end{aligned}$$

Now let's see the properties of this technique. The system is not resilient, because it is not fully immune to distortive attacks, i.e. $\text{DenSem} \not\sqsubseteq \prod \{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$.

PROOF. Suppose that $\text{DenSem} \sqsubseteq \prod \overline{\mathfrak{M}}$, so $\forall X \in \wp(\Sigma^+)$ it holds that $\text{DenSem}(X) \subseteq \prod \overline{\mathfrak{M}}(X)$. Let $X = \mathcal{W}_s$, for a generic signature s , so $\prod \overline{\mathfrak{M}}(X) = \mathcal{W}_s$. But $\mathcal{W}_s \subsetneq \text{DenSem}(\mathcal{W}_s)$, because there is for sure at least a trace with the same initial and final state of a trace in \mathcal{W}_s but with $\text{Branch}(\sigma) \neq \text{Bin}(s)$. For example, take a program equals to $\mathfrak{M}(s)$ but in which is inserted an opaque predicate. Clearly its traces are in $\text{DenSem}(\mathcal{W}_s)$ but they aren't in \mathcal{W}_s , because this traces have a different number of conditional statements. So there is a X for which $\text{DenSem}(X) \not\subseteq \prod \overline{\mathfrak{M}}(X)$ and hence $\text{DenSem} \not\sqsubseteq \prod \overline{\mathfrak{M}}$. \square

Indeed the system is vulnerable to control flow obfuscation techniques. For example, an edge-flipping attack and an opaque predicate insertion attack are able to damage the stegomark. Like showed in Section 2.4, the most powerful \top -secret attacker for path-based watermarking is

$$\mathcal{K}_{\mathcal{L},(\text{id}),\top}(\text{id}) = \{X \in \wp(\Sigma^+) \mid P \in \mathcal{P}, X = \bigcup_{Q \in \mathcal{Q}} (\mathcal{L}(P, Q))_+\} \cup \{\Sigma^+\}$$

and it abstracts in the same object the traces of all possible stegoprograms related to the same cover program. Finally the system is also $\alpha_{\mathcal{O}}^{\text{out}}$ accurate, indeed the behavior of the cover program is preserved w.r.t. this observation. Clearly for every program P and for every signature s it holds that $\alpha_{\mathcal{O}}^{\text{out}}(\llbracket P \rrbracket_{Den}) = \alpha_{\mathcal{O}}^{\text{out}}(\llbracket P_s \rrbracket_{Den})$. In fact the embedding algorithm guarantee that the insertion/modification of the conditional statements doesn't affect the variables of the cover program. So, with the same input, the cover program and the stegoprogram give the same output.

4.3 Abstract constant propagation watermarking

Abstract software watermarking technique, conceived by *Cousot and Cousot* [8], where the signature (a natural number) is inserted into a particular variable which, although being modified during program execution, it remains constant modulo an integer n (after being initialized). Clearly the insertion of the signature doesn't alter the semantics of the cover program. In practice, even if the programming languages imposes a maximum limit for integers, the signature can be arbitrarily large, provided that it is decomposed into k parts, using the Chinese remainder theorem. In this case the watermark is the composition of k partial watermarks which encode each one a number smaller than the limit imposed by programming language. In the following, it is assumed that such limit doesn't exist and the signature is inserted without decomposed it.

In the original program it is declared a new variable w which have to hide the value of s . Then two integer-valued polynomials `init` and `iter` are chosen for, respectively, to initialize and to modify w . The instruction `w := init(1)` (initialization) is inserted into a random point in the program, but on condition that such a point always runs, while instruction `w := iter(w)` (iteration) is inserted into a random point the (after the initialization). The polynomials must satisfy the following conditions⁹: `init(1) = s mod n` and `iter(w) = s mod n`. Therefore, once initialized, w remains constant modulo n , even if its value changes in \mathbb{Z} in each iteration.

For the formalization of this technique in the framework, we have to extend the information contained in the states of execution traces. So, assuming to have an enumeration of the programs in IMP , we insert in the state the

⁹For the generation of such polynomial one can take advantage of the Horner method [8]

identifier of the program which contains it, i.e. $\Sigma = \langle \text{Com} \times \text{Con} \times \mathbb{N} \rangle$. For short, we write σ^i the identifier contained in the states of the trace σ . Now we can define the relation $\approx \subseteq \Sigma^+ \times \Sigma^+$ such that for all $\sigma, \bar{\sigma}$ we have $\sigma \approx \bar{\sigma}$ iff $\sigma^i = \bar{\sigma}^i$. It is straightforward to note that \approx is an equivalence relation. So, given a set of traces X we indicate with X/\approx its quotient set, i.e. the set of its equivalence class induced by \approx . Let \mathbb{Z}_n be the (quotient) ring of integers modulo n and $\bar{\mathbb{Z}}_n \stackrel{\text{def}}{=} \{\perp, \mathbb{Z}_n\} \cup \mathbb{Z}_n$. We assume to have a function $\text{IsConst}_n : \wp(\Sigma^+) \times \text{Lab} \rightarrow (\text{Var} \rightarrow \bar{\mathbb{Z}}_n)$ such that: given a set of traces X and a label l , $\text{IsConst}_n(X, l)(y)$ returns the value of y modulo n if the variable is constant in \mathbb{Z}_n into the set of traces X at label l . If the variable is undefined it returns \perp and \mathbb{Z}_n if the variable is not constant modulo n ¹⁰. Finally $\text{Const}_n : \wp(\Sigma^+) \rightarrow \wp(\mathbb{Z}_n)$ is defined as:

$$\text{Const}_n \stackrel{\text{def}}{=} \lambda X . \bigcup_{y \in \text{Var}} \left\{ \text{IsConst}_n(X, l)(y) \mid \begin{array}{l} \exists l \in \text{Lab} . \\ \text{IsConst}_n(X, l)(y) \in \mathbb{Z}_n \end{array} \right\}$$

In short, this function returns all the values in \mathbb{Z}_n of the variables that are constant modulo n into a set of traces (at some label). The semantics $\langle P \rangle_+^\beta$ performs constant propagation modulo n for program P , so

$$\beta = \left\{ X \in \wp(\Sigma^+) \mid \begin{array}{l} \exists \mathcal{N} \subseteq \mathbb{Z}_n . \mathcal{N} \neq \emptyset \wedge \\ X = \bigcup_{id \in \mathbb{N}} \left\{ Y \in \wp(\Sigma^+) \mid \begin{array}{l} \forall \sigma \in Y . \sigma^i = id \wedge \\ \text{Const}_n(Y) = \mathcal{N} \end{array} \right\} \end{array} \right\} \cup \{\Sigma^+\}$$

The domain β contains all the sets of traces with the same values of the variables constant modulo n . With $\mathcal{W}_s \stackrel{\text{def}}{=} \bigcup_{id \in \mathbb{N}} \{X \in \wp(\Sigma^+) \mid \forall \sigma \in X . \sigma^i = id \wedge \text{Const}_n(X) = \{s \bmod n\}\}$ we indicate the set of traces which have a constant variable that encode the signature s . This is an abstract technique so $\eta = \text{id}$. Clearly $\langle P \rangle_+^\beta = \mathcal{W}_s$ and so $\mathfrak{M}(s) = \{\mathcal{W}_s, \Sigma^+\}$. The domain β can be defined as $\beta \stackrel{\text{def}}{=} \beta_\gamma \circ \beta_\alpha$ where $\beta_\alpha : \wp(\Sigma^+) \rightarrow \wp(\mathbb{Z}_n)$ and $\beta_\gamma : \mathbb{Z}_n \rightarrow \wp(\Sigma^+)$ are

$$\beta_\alpha \stackrel{\text{def}}{=} \lambda X . \begin{cases} \emptyset & \text{if } X = \emptyset \\ \mathcal{N} & \text{if } \mathcal{N} \subsetneq \mathbb{Z}_n \wedge \mathcal{N} \neq \emptyset \wedge \forall X_j \in X/\approx . \text{Const}_n(X_j) = \mathcal{N} \\ \mathbb{Z}_n & \text{otherwise} \end{cases}$$

$$\beta_\gamma \stackrel{\text{def}}{=} \lambda \mathcal{N} . \begin{cases} \emptyset & \text{if } \mathcal{N} = \emptyset \\ \bigcup_{id \in \mathbb{N}} \left\{ X \in \wp(\Sigma^+) \mid \begin{array}{l} \forall \sigma \in X . \sigma^i = id \wedge \\ \text{Const}_n(X) = \mathcal{N} \end{array} \right\} & \text{if } \mathcal{N} \notin \{\emptyset, \mathbb{Z}_n\} \\ \Sigma^+ & \text{otherwise} \end{cases}$$

¹⁰This function implements a constant propagation analysis

Instead $\overline{\mathfrak{M}}(s)$ can be defined as $\overline{\mathfrak{M}}(s) \stackrel{\text{def}}{=} \overline{\mathfrak{M}}(s)_\gamma \circ \overline{\mathfrak{M}}(s)_\alpha$ where $\overline{\mathfrak{M}}(s)_\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ and $\overline{\mathfrak{M}}(s)_\gamma : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ are

$$\overline{\mathfrak{M}}(s)_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} \emptyset & \text{if } X = \emptyset \\ \mathcal{W}_s & \text{if } X \subseteq \mathcal{W}_s \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \overline{\mathfrak{M}}(s)_\gamma \stackrel{\text{def}}{=} \lambda X. X$$

It is simple to see that for all signature s we have $\overline{\mathfrak{M}}(s)(\wp(\Sigma^+)) \subseteq \beta(\wp(\Sigma^+))$ and so $\beta \sqsubseteq \overline{\mathfrak{M}}(s)$. Then we have to check if \mathfrak{L} is a stegoembedder w.r.t. β . The input domain is id so we have to check if for all set of states X we have

$$\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta(X) \wedge \wp(\mathfrak{M}(s))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta(X)$$

Clearly for every possible set of initial states, the constant propagation modulo n of $\mathfrak{L}(P, \mathfrak{M}(s))$ is the same, i.e. it exists $\mathcal{N} \subsetneq \mathbb{Z}_n$ not empty such that $\text{Const}_n(\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta) = \mathcal{N}$. For how the technique is designed, \mathcal{N} is the set $\{s \bmod n\}$. So we have that $\wp(\mathfrak{L}(L, \mathfrak{M}(s)))_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states. Now, the constant propagation modulo n of $\mathfrak{M}(s)$ is exactly $\{s \bmod n\}$, so $\wp(\mathfrak{M}(s))_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states. Therefore we have $\forall X \in \wp(\Sigma)$.

$$\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta(X)$$

We can also note that, for every signature s , $\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta$ is \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. In fact we have:

$$\begin{aligned} \wp(P_s)_+^\beta(X) \circ \eta &= \overline{\mathfrak{M}}(s) \circ \wp(P_s)_+^\beta(X) \circ \eta \\ \mathcal{W}_s &= \overline{\mathfrak{M}}(s)(\mathcal{W}_s) \\ \mathcal{W}_s &= \mathcal{W}_s \end{aligned}$$

Now let's see the properties of this technique. The credibility of the system is high because, for all signature s , $\overline{\mathfrak{M}}(s)$ is the atomic closure of $\wp(\mathfrak{M}(s))_+^\beta$. The system is not resilient, because it is not fully immune to distortive attacks, i.e. $\text{DenSem} \not\sqsubseteq \sqcap \{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$.

PROOF. Suppose that $\text{DenSem} \sqsubseteq \sqcap \overline{\mathfrak{M}}$, so $\forall X \in \wp(\Sigma^+)$ it holds that $\text{DenSem}(X) \subseteq \sqcap \overline{\mathfrak{M}}(X)$. Let $X = \mathcal{W}_s$, for a generic signature s , so $\sqcap \overline{\mathfrak{M}}(X) = \mathcal{W}_s$. But $\mathcal{W}_s \subsetneq \text{DenSem}(\mathcal{W}_s)$, because there is for sure at least a trace with the same initial and final state of a trace in \mathcal{W}_s which belongs to a program that doesn't have a constant variable modulo n equal to s . So there is a X for which $\text{DenSem}(X) \not\subseteq \sqcap \overline{\mathfrak{M}}(X)$ and hence $\text{DenSem} \not\sqsubseteq \sqcap \overline{\mathfrak{M}}$. \square

Like showed in Section 2.4, the most powerful \top -secret attacker for abstract constant propagation watermarking is

$$\mathcal{K}_{\mathfrak{L}, (\text{id}), \top}(\text{id}) = \{X \in \wp(\Sigma^+) \mid P \in \mathcal{P}, X = \bigcup_{Q \in \mathcal{Q}} \wp(\mathfrak{L}(P, Q))_+ \} \cup \{\Sigma^+\}$$

and it abstracts in the same object the traces of all possible stegoprograms related to the same cover program. Finally the system is also $\alpha_{\mathcal{O}}^{out}$ accurate, indeed the behavior of the cover program is preserved w.r.t. this observation. Clearly for every program P and for every signature s it holds that $\alpha_{\mathcal{O}}^{out}(\langle P \rangle_{Den}) = \alpha_{\mathcal{O}}^{out}(\langle P_s \rangle_{Den})$. In fact the code inserted by the stegoembedder doesn't affect the variables of the cover program (the added code just add new variables and let untouched the variables of the cover program). So the output variables of the cover program are the same of the output variables of the stegoprogram.

4.4 Block-reordering watermarking

Static software watermarking technique, conceived by *Davidson and Myhrvold* [12], where the signature (a natural number) is codified as a permutation of the basic blocks of CFG¹¹ of the cover program. In particular a program, whose CFG is equal to the graph which encode the signature, is generated and then its instructions are substituted with the instructions of the cover program, in a way that its semantics remains unmodified.

Let $\mathcal{E} : \mathbb{N} \rightarrow \mathbb{G}$ be a function that codify a signature in a graph. Let $\text{CFG} : \Sigma^+ \rightarrow \mathbb{G}$ be a function that, given a trace σ , outputs the CFG of σ ¹². The semantics $\langle P \rangle_+^\beta$ extracts the CFG of P , so the extraction domain β is:

$$\beta \stackrel{\text{def}}{=} \{X \in \wp(\Sigma^+) \mid \exists g \in \mathbb{G}. X = \{\sigma \in \Sigma^+ \mid \text{CFG}(\sigma) = g\}\} \cup \{\Sigma^+\}$$

So in β there are all the sets of traces with the same CFG. With $\mathcal{W}_s \stackrel{\text{def}}{=} \{\sigma \in \Sigma^+ \mid \mathcal{E}(s) = \text{CFG}(\sigma)\}$ we indicate the set of traces whose CFG codifies the signature s . This is a static technique so $\eta = \text{id}$. Clearly $\langle \mathfrak{M}(s) \rangle_+^\beta = \mathcal{W}_s$ and so $\overline{\mathfrak{M}}(s) = \{\mathcal{W}_s, \Sigma^+\}$. Let $\overline{\mathbb{G}} \stackrel{\text{def}}{=} \{\perp, \mathbb{G}\} \cup \{\mathbb{G}\}$. The domain β can be defined as $\beta \stackrel{\text{def}}{=} \beta_\gamma \circ \beta_\alpha$ where $\beta_\alpha : \wp(\Sigma^+) \rightarrow \overline{\mathbb{G}}$ and $\beta_\gamma : \overline{\mathbb{G}} \rightarrow \wp(\Sigma^+)$ are

$$\beta_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} \perp & \text{if } X = \emptyset \\ g & \text{if } \forall \sigma \in X. g = \text{CFG}(\sigma) \\ \mathbb{G} & \text{otherwise} \end{cases}$$

$$\beta_\gamma \stackrel{\text{def}}{=} \lambda g. \begin{cases} \emptyset & \text{if } g = \perp \\ \{\sigma \in \Sigma^+ \mid g = \text{CFG}(\sigma)\} & \text{if } g \in \mathbb{G} \\ \Sigma^+ & \text{otherwise} \end{cases}$$

Instead $\overline{\mathfrak{M}}(s)$ can be defined as $\overline{\mathfrak{M}}(s) \stackrel{\text{def}}{=} \overline{\mathfrak{M}}(s)_\gamma \circ \overline{\mathfrak{M}}(s)_\alpha$ where $\overline{\mathfrak{M}}(s)_\alpha :$

¹¹Control Flow Graph

¹²Building the CFG is easily implementable analyzing a program trace

$\wp(\Sigma^+) \longrightarrow \wp(\Sigma^+)$ and $\overline{\mathfrak{M}}(s)_\gamma : \wp(\Sigma^+) \longrightarrow \wp(\Sigma^+)$ are

$$\overline{\mathfrak{M}}(s)_\alpha \stackrel{\text{def}}{=} \lambda X. \begin{cases} \emptyset & \text{if } X = \emptyset \\ \mathcal{W}_s & \text{if } X \subseteq \mathcal{W}_s \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \overline{\mathfrak{M}}(s)_\gamma \stackrel{\text{def}}{=} \lambda X. X$$

It is simple to see that for all signature s we have $\overline{\mathfrak{M}}(s)(\wp(\Sigma^+)) \subseteq \beta(\wp(\Sigma^+))$ and so $\beta \sqsubseteq \overline{\mathfrak{M}}(s)$. Then we have to check if \mathfrak{L} is a stegoembedder w.r.t. β . The input domain is id so we have to check if for all set of states X we have

$$\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta(X) \wedge \wp(\mathfrak{M}(s))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta$$

Clearly for every possible set of initial states, the CFG of $\mathfrak{L}(P, \mathfrak{M}(s))$ is the same, i.e. it exists $g \in \mathbb{G}$ such that $\forall \sigma \in \wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+$ we have $g = \text{CFG}(\sigma)$. For how the technique is designed, this graph is equal to $\mathcal{E}(s)$, by design. So we have that $\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states. Now, the CFG of $\mathfrak{M}(s)$ is also $\mathcal{E}(s)$, so $\wp(\mathfrak{M}(s))_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states. Therefore we have $\forall X \in \wp(\Sigma)$.

$$\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta(X) = \wp(\mathfrak{M}(s))_+^\beta$$

We can also note that, for every signature s , $\wp(\mathfrak{L}(P, \mathfrak{M}(s)))_+^\beta$ is \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. In fact we have:

$$\begin{aligned} \wp(P_s)_+^\beta(X) \circ \eta &= \overline{\mathfrak{M}}(s) \circ \wp(P_s)_+^\beta(X) \circ \eta \\ \mathcal{W}_s &= \overline{\mathfrak{M}}(s)(\mathcal{W}_s) \\ \mathcal{W}_s &= \mathcal{W}_s \end{aligned}$$

Now let's see the properties of this technique. The system is not resilient, because it is not fully immune to distortive attacks, i.e. $\text{DenSem} \not\sqsubseteq \prod \{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$.

PROOF. Suppose that $\text{DenSem} \sqsubseteq \prod \overline{\mathfrak{M}}$, so $\forall X \in \wp(\Sigma^+)$ it holds that $\text{DenSem}(X) \subseteq \prod \overline{\mathfrak{M}}(X)$. Let $X = \mathcal{W}_s$, for a generic signature s , so $\prod \overline{\mathfrak{M}}(X) = \mathcal{W}_s$. But $\mathcal{W}_s \subsetneq \text{DenSem}(\mathcal{W}_s)$, because there is for sure at least a trace with the same initial and final state of a trace in \mathcal{W}_s but with $\text{CFG}(\sigma) \neq \mathcal{E}(s)$. For example, take a program equals to $\mathfrak{M}(s)$ but in which is inserted an opaque predicate. Clearly its traces are in $\text{DenSem}(\mathcal{W}_s)$ but they aren't in \mathcal{W}_s , because this traces have a different CFG. So there is a X for which $\text{DenSem}(X) \not\subseteq \prod \overline{\mathfrak{M}}(X)$ and hence $\text{DenSem} \not\sqsubseteq \prod \overline{\mathfrak{M}}$. \square

Indeed the system is vulnerable to control flow obfuscation techniques (basically the ones which modify the CFG). For example, a CFG flattening attack is able to damage the stegoemark. Like showed in Section 2.4, the most powerful \top -secret attacker for block-reordering watermarking is

$$\mathcal{K}_{\mathfrak{L}, (\text{id}), \top}(\text{id}) = \{X \in \wp(\Sigma^+) \mid P \in \mathbb{P}, X = \bigcup_{Q \in \mathbb{Q}} \wp(\mathfrak{L}(P, Q))_+ \} \cup \{\Sigma^+\}$$

and it abstracts in the same object the traces of all possible stegoprograms related to the same cover program. Finally the system is also $\alpha_{\mathcal{O}}^{out}$ accurate, indeed the behavior of the cover program is preserved w.r.t. this observation. Clearly for every program P and for every signature s it holds that $\alpha_{\mathcal{O}}^{out}(\llbracket P \rrbracket_{Den}) = \alpha_{\mathcal{O}}^{out}(\llbracket P_s \rrbracket_{Den})$. In fact the reordering of nodes of the CFG doesn't affect the variables of the cover program (there are new variables in the stegoprogram but, by design, these don't interfere with the variables of the cover program). So, with the same input, the cover program and the stegoprogram give the same output.

4.5 Dynamic graph-based watermarking

Dynamic software watermarking technique, conceived by *Collberg and Thomborson* [2], where the signature (a natural number) is encoded by a graph allocated in the dynamic memory (in the *heap*) of the program, during a particular execution of the program. This execution is generated by a particular sequence of input values I_0, I_1, \dots, I_k called enabling sequence. Given a signature, a graph that encode the signature and the code that build this graph are generated. The embedder takes the program and it adds the code that generates the parts of the graph in some locations of the cover program, but this code is executed only with the enabling input. Furthermore the code that builds the root of the graph is executed at last, so the root is the last node of the graph that is inserted in the heap (this is done in order to recognize the right graph in the heap at the extraction phase). With ϵ we indicate the empty sequence. Let $\mathcal{E} : \mathbb{N} \rightarrow \mathbb{G}$ be a function that codify a signature in a graph and $\text{Heap} : \mathbb{H} \rightarrow \wp(\mathbb{G})$ be a function that extracts the graphs memorized in an heap. For the formalization of this technique in the framework, we have to extend the information contained in the states of execution traces. So we insert in the state the heap of the program at the current state, i.e. $\Sigma = \langle \text{Com} \times \text{Con} \times \mathbb{H} \rangle$. So $\sigma = \langle C, \zeta, \mathcal{H} \rangle$, where $\mathcal{H} \in \mathbb{H}$ is an heap. Let $\mathcal{E} : \mathbb{G} \rightarrow \wp(\Sigma^+)$ ¹³ the function:

$$\mathcal{E} \stackrel{\text{def}}{=} \lambda g. \left\{ \sigma \in \Sigma^+ \left| \begin{array}{l} |\sigma| = n + 1, \sigma_n = \langle C_n, \langle \rho_n, \iota_n \rangle, \mathcal{H}_n \rangle, \\ \text{top}(\iota_n) = \epsilon, g \in \text{Heap}(\mathcal{H}_n), \text{root}(g) \in \text{Heap}(\mathcal{H}_n) \\ \forall j \in [0, n). \text{root}(g) \notin \text{Heap}(\mathcal{H}_j) \end{array} \right. \right\}$$

The semantics $\llbracket P \rrbracket_+^\beta$ extracts the graph (with the root inserted at last) memorized in the heap of the program P when all the input values are consumed, so the domain β is

$$\beta \stackrel{\text{def}}{=} \{ X \in \wp(\Sigma^+) \mid g \in \mathbb{G}, X = \mathcal{E}(g) \} \cup \{ \Sigma^+ \}$$

and it contains all the sets of traces which have the same graph (with the root inserted at last) memorized in the heap, when all the input values are

¹³This function is not necessary, it is only useful for a simpler reading of the text

consumed. With $\mathcal{W}_s = \mathcal{E}(\mathcal{E}(s))$ we indicate the set of traces for which, when all the input values are consumed, the heap contains the encoding of the signature s . This is a dynamic technique, so $\eta = \wp(\mathcal{I}) \cup \{\Sigma^+\}$, where \mathcal{I} represents the set of states enabling the watermark, i.e.

$$\mathcal{I} \stackrel{\text{def}}{=} \left\{ \varsigma \in \Sigma \mid \begin{array}{l} \varsigma = \langle C, \langle \rho, \iota \rangle, \mathcal{H} \rangle, |\iota| = |I|, \\ \forall j \in [0, |I|) . \text{top}(\text{next}(\iota)^j) = I_j \end{array} \right\}$$

The domain η can be defined as $\eta \stackrel{\text{def}}{=} \eta_\gamma \circ \eta_\alpha$ where $\eta_\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ and $\eta_\gamma : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ are

$$\eta_\alpha \stackrel{\text{def}}{=} \lambda X . \begin{cases} X & \text{if } X \subseteq \mathcal{I} \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \eta_\gamma \stackrel{\text{def}}{=} \lambda X . X$$

Clearly $(\overline{\mathfrak{M}}(s))_+^\beta = \mathcal{W}_s$ and so $\overline{\mathfrak{M}}(s) = \{\mathcal{W}_s, \Sigma^+\}$. Let $\overline{\mathbb{G}} \stackrel{\text{def}}{=} \{\perp, \mathbb{G}\} \cup \{\mathbb{G}\}$. The domain β can be defined as $\beta \stackrel{\text{def}}{=} \beta_\gamma \circ \beta_\alpha$ where $\beta_\alpha : \wp(\Sigma^+) \rightarrow \overline{\mathbb{G}}$ and $\beta_\gamma : \overline{\mathbb{G}} \rightarrow \wp(\Sigma^+)$ are

$$\beta_\alpha \stackrel{\text{def}}{=} \lambda X . \begin{cases} \perp & \text{if } X = \emptyset \\ g \in \mathbb{G} & \text{if } \forall \sigma \in X . \begin{array}{l} |\sigma| = n + 1, \sigma_n = \langle C_n, \langle \rho_n, \iota_n \rangle, \mathcal{H}_n \rangle \\ g \in \text{Heap}(\mathcal{H}_n), \text{root}(g) \in \text{Heap}(\mathcal{H}_n) \\ \forall j \in [0, n) . \text{root}(g) \notin \text{Heap}(\mathcal{H}_j), \text{top}(\iota_n) = \epsilon \end{array} \\ \mathbb{G} & \text{otherwise} \end{cases}$$

$$\beta_\gamma \stackrel{\text{def}}{=} \lambda g . \begin{cases} \emptyset & \text{if } g = \perp \\ \mathcal{E}(g) & \text{if } g \in \mathbb{G} \\ \Sigma^+ & \text{otherwise} \end{cases}$$

Instead $\overline{\mathfrak{M}}(s)$ can be defined as $\overline{\mathfrak{M}}(s) \stackrel{\text{def}}{=} \overline{\mathfrak{M}}(s)_\gamma \circ \overline{\mathfrak{M}}(s)_\alpha$ where $\overline{\mathfrak{M}}(s)_\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ and $\overline{\mathfrak{M}}(s)_\gamma : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$ are

$$\overline{\mathfrak{M}}(s)_\alpha \stackrel{\text{def}}{=} \lambda S . \begin{cases} \mathcal{W}_s & \text{if } S \subseteq \mathcal{W}_s \\ \Sigma^+ & \text{otherwise} \end{cases} \quad \overline{\mathfrak{M}}(s)_\gamma \stackrel{\text{def}}{=} \lambda X . X$$

It is simple to see that for all signature s we have $\overline{\mathfrak{M}}(s)(\wp(\Sigma^+)) \subseteq \beta(\wp(\Sigma^+))$ and so $\beta \sqsubseteq \overline{\mathfrak{M}}(s)$. Then we have to check if \mathfrak{L} is a stegoembedder w.r.t. β . The input domain is not id so we have to check if for all set of states X we have

- ♠ $X \in \eta(\Sigma) \Rightarrow \{\mathfrak{L}(P, \mathfrak{M}(s))\}_+^\beta(X) = \{\mathfrak{M}(s)\}_+^\beta(X) \wedge \{\mathfrak{M}(s)\}_+^\beta(X) = \{\mathfrak{M}(s)\}_+^\beta(X)$
- ♣ $X \notin \eta(\Sigma) \Rightarrow \{\mathfrak{L}(P, \mathfrak{M}(s))\}_+^\beta(X) = \{\mathfrak{P}\}_+^\beta(X)$

If $X \in \eta(\Sigma)$ then $X \subseteq \mathcal{I}$. All such X contain states which encode the enabling input, so the $\mathfrak{L}(P, \mathfrak{M}(s))$ executes the code which builds the graph

$\mathcal{E}(s)$ in the heap. So we have that $\llbracket \mathcal{L}(L, \mathfrak{M}(s)) \rrbracket_+^\beta(X) = \mathcal{W}_s$ for every possible set of initial states that satisfy η . Now, the same reasoning can be done for $\mathfrak{M}(s)$, because it codify the signature by design (starting from the sets of input states which encode the enabling input). So $\llbracket \mathfrak{M}(s) \rrbracket_+^\beta(X) = \mathcal{W}_s$ for every $X \in \eta(\Sigma)$. Therefore we have that \spadesuit holds. If $X \notin \eta(\Sigma)$ then $X \not\subseteq \mathcal{I}$. All such X don't contain states which encode the enabling input, so the $\mathcal{L}(P, \mathfrak{M}(s))$ doesn't execute the code which build $\mathcal{E}(s)$ in the heap. So the objects built in the heap are with high probability the same of those built by P . Even better, it is very likely that both $\llbracket \mathcal{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta(X)$ and $\llbracket P \rrbracket_+^\beta$ are equal to Σ^+ . Indeed we have that \clubsuit holds.

We can also note that, for every signature s , $\llbracket \mathcal{L}(P, \mathfrak{M}(s)) \rrbracket_+^\beta$ is \mathcal{F} -complete for η and $\overline{\mathfrak{M}}(s)$. In fact we have, when $X \in \eta(\Sigma)$:

$$\begin{aligned} \llbracket P_s \rrbracket_+^\beta(X) \circ \eta &= \overline{\mathfrak{M}}(s) \circ \llbracket P_s \rrbracket_+^\beta(X) \circ \eta \\ \mathcal{W}_s &= \overline{\mathfrak{M}}(s)(\mathcal{W}_s) \\ \mathcal{W}_s &= \mathcal{W}_s \end{aligned}$$

Now let's see the properties of this technique. The system is not resilient, because it is not fully immune to distortive attacks, i.e. $\text{DenSem} \not\sqsubseteq \prod \{\overline{\mathfrak{M}}(s) \mid s \in \mathcal{S}\}$.

PROOF. Suppose that $\text{DenSem} \sqsubseteq \prod \overline{\mathfrak{M}}$, so $\forall X \in \wp(\Sigma^+)$ it holds that $\text{DenSem}(X) \subseteq \prod \overline{\mathfrak{M}}(X)$. Let $X = \mathcal{W}_s$, for a generic signature s , so $\prod \overline{\mathfrak{M}}(X) = \mathcal{W}_s$. But $\mathcal{W}_s \subsetneq \text{DenSem}(\mathcal{W}_s)$, because there is for sure at least a trace with the same initial and final state of a trace in \mathcal{W}_s but without $\mathcal{E}(s)$ among the objects memorized in the heap. For example, take a program equals to $\mathfrak{M}(s)$ but in which the code that inserts the root node is duplicated. Clearly the traces of this program are in $\text{DenSem}(\mathcal{W}_s)$ but they aren't in \mathcal{W}_s , because in this traces not only the last heap contains the root of the graph. So there is a X for which $\text{DenSem}(X) \not\subseteq \prod \overline{\mathfrak{M}}(X)$ and hence $\text{DenSem} \not\sqsubseteq \prod \overline{\mathfrak{M}}$. \square

Indeed the system is vulnerable to techniques that modifies the structure of the runtime objects created. For example, a node-splitting attack is able to damage the stegomark (if it modifies the structure of the root node of the graph then the extractor is not able to recognize the stegomark). Like showed in Section 2.4, the most powerful \top -secret attacker for dynamic graph-based watermarking is

$$\mathcal{K}_{\mathcal{E},(\text{id}),\top(\text{id})} = \{X \in \wp(\Sigma^+) \mid P \in \mathcal{P}, X = \bigcup_{Q \in \mathcal{Q}} \llbracket \mathcal{L}(P, Q) \rrbracket_+ \} \cup \{\Sigma^+\}$$

and it abstracts in the same object the traces of all possible stegoprograms related to the same cover program.

Finally the system is also $\alpha_{\mathcal{O}}^{\text{out}}$ accurate, indeed the behavior of the cover program is preserved w.r.t. this observation. Clearly for every program P and for every signature s it holds that $\alpha_{\mathcal{O}}^{\text{out}}(\llbracket P \rrbracket_{\text{Den}}) = \alpha_{\mathcal{O}}^{\text{out}}(\llbracket P_s \rrbracket_{\text{Den}})$. In fact

the embedding algorithm guarantee that the code inserted doesn't affect the variables of the cover program. So, with the same input, the cover program and the stegoprogram give the same output.

5 Comparison

Until now we have formalized and validated the framework. The next step is clearly to show how to use this tool for performing comparisons between different software watermarking systems, in order to be able to chose a technique rather than another according to the needs. For example, if we are interested in resilience so we can compare two systems to determine which is the best, regard this property.

5.1 Resilience

In Section 3.2 we have discussed how different watermarking systems $\mathfrak{A}_1 = \langle \mathfrak{L}_1, \mathfrak{M}_1, \beta_1 \rangle$ and $\mathfrak{A}_2 = \langle \mathfrak{L}_2, \mathfrak{M}_2, \beta_2 \rangle$ could be compared w.r.t. resilience by comparing the degree of abstraction of $\prod\{\overline{\mathfrak{M}}_1(s) \mid s \in \mathcal{S}\}$ and $\prod\{\overline{\mathfrak{M}}_2(s) \mid s \in \mathcal{S}\}$. Of course it may happen that these two abstractions are not comparable. In this case what we can do is to compare their resilience w.r.t. a specific distortiv attack.

In this context we consider an attacker every possible program transformation. In fact there are specific programs used with the purpose to defeat software watermarking, the "real" attackers¹⁴, but also there are programs which accidentally can damage the watermark. For example, most code optimization techniques could interfere with software watermarking. Like in [10] we model the attackers as semantic program transformations, considering that their syntactic counterpart can always be derived. Indeed, any syntactic program transformer t , altering the code of P and returning a new program P' , induces a corresponding semantic transformer \bar{t} turning $\langle P \rangle_+$ into $\langle P' \rangle_+$ [9].

5.1.1 Edge-flipping

The edge-flipping obfuscation inverts the branches of every conditional statement of a program, namely it exchanges the code executed in the true branch with the code executed in the false branch, for every conditional command. In order to preserve the program semantics, the obfuscation also have to replace every branch condition, called *guard*, with its negate.

¹⁴In this category we found the obfuscation techniques

The semantic transformation $\mathfrak{t}^{\text{ef}} : \wp(\Sigma^+) \longrightarrow \wp(\Sigma^+)$ is defined as:

$$\begin{aligned} \overline{\mathfrak{t}^{\text{ef}}}(X) &\stackrel{\text{def}}{=} \{\overline{\mathfrak{t}^{\text{ef}}}(\sigma) \mid \sigma \in X\} \\ \overline{\mathfrak{t}^{\text{ef}}}(\langle C, \zeta \rangle \sigma) &\stackrel{\text{def}}{=} \begin{cases} \langle C, \zeta \rangle \overline{\mathfrak{t}^{\text{ef}}}(\sigma) & \text{if } C = L : \text{stop}; \vee \\ & C = L : A \rightarrow L'; \\ \langle L : \neg B \rightarrow \{L_F, L_T\};, \zeta \rangle \overline{\mathfrak{t}^{\text{ef}}}(\sigma) & \text{if } C = L : B \rightarrow \{L_T, L_F\}; \end{cases} \end{aligned}$$

The most concrete preserved property is

$$\delta_{\mathfrak{t}^{\text{ef}}} = \bigsqcup_{P \in \text{IMP}} \{X \subseteq \Sigma^+ \mid \text{Pres}_{P, \overline{\mathfrak{t}^{\text{ef}}}}(X)\}$$

where $\text{Pres}_{P, \overline{\mathfrak{t}^{\text{ef}}}}(X)$ if and only if:

$$\forall Y \subseteq \langle P \rangle_+. Y \subseteq X \Rightarrow \bigcup \{Z \subseteq \Sigma^+ \mid Z = \overline{\mathfrak{t}^{\text{ef}}}(Y)\} \subseteq X$$

This means that a set of traces X is preserved by edge-flipping transformation if it contains all the traces that can be obtained from traces in X by inverting every conditional branch and negating the related guard.

5.1.2 Opaque predicate insertion

Let $\mathcal{I} : \text{IMP} \longrightarrow \wp(\text{Lab})$ be the result of a preliminary static analysis that given a program returns the subset of its labels where it is possible to insert opaque predicates. Usually the preliminary static analysis consists of a combination of liveness analysis and static analysis. Given a program P , we assume to know the set $\mathcal{I}^P \subseteq \text{lab} \llbracket P \rrbracket$ of labels that the preliminary static analysis has classified as candidate for opaque predicate insertion. Let P^t a true opaque predicate, i.e. a boolean expression that is always evaluated to tt, let \hat{L} a co-label of P , i.e. $\hat{L} \notin \text{lab} \llbracket P \rrbracket$, and let \tilde{L} a random label of P . We write with $\mathbf{lab}(\varsigma)$ the label of the command contained in the state ς , i.e. if $\varsigma = \langle C, \zeta \rangle$ then $\mathbf{lab}(\varsigma) = \text{lab} \llbracket C \rrbracket$.

The semantic transformation $\overline{\mathfrak{t}^{\text{opi}}} : \wp(\Sigma^+) \times \wp(\text{Lab}) \longrightarrow \wp(\Sigma^+)$ is defined as:

$$\overline{\mathfrak{t}^{\text{opi}}}(X, \mathcal{I}^P) \stackrel{\text{def}}{=} \{\overline{\mathfrak{t}^{\text{opi}}}(\sigma, \mathcal{I}^P) \mid \sigma \in X\}$$

$$\overline{\mathfrak{t}^{\text{opi}}}(\langle C, \zeta \rangle \sigma, \mathcal{I}^P) \stackrel{\text{def}}{=} \begin{cases} \langle C, \zeta \rangle \overline{\mathfrak{t}^{\text{opi}}}(\sigma, \mathcal{I}^P) & \text{if } \mathbf{lab}(\langle C, \zeta \rangle) \notin \mathcal{I}^P \\ \langle L : P^t \rightarrow \{\hat{L}, \tilde{L}\};, \zeta \rangle \langle \hat{L} : A \rightarrow L';, \zeta \rangle \overline{\mathfrak{t}^{\text{opi}}}(\sigma, \mathcal{I}^P) & \text{if } \mathbf{lab}(\langle C, \zeta \rangle) \in \mathcal{I}^P \wedge \\ & C = L : A \rightarrow L'; \\ \langle L : P^t \rightarrow \{\hat{L}, \tilde{L}\};, \zeta \rangle \langle \hat{L} : B \rightarrow \{L_T, L_F\};, \zeta \rangle \overline{\mathfrak{t}^{\text{opi}}}(\sigma, \mathcal{I}^P) & \text{if } \mathbf{lab}(\langle C, \zeta \rangle) \in \mathcal{I}^P \wedge \\ & C = L : B \rightarrow \{L_T, L_F\}; \\ \langle L : P^t \rightarrow \{\hat{L}, \tilde{L}\};, \zeta \rangle \langle \hat{L} : \text{stop};, \zeta \rangle \overline{\mathfrak{t}^{\text{opi}}}(\sigma, \mathcal{I}^P) & \text{if } \mathbf{lab}(\langle C, \zeta \rangle) \in \mathcal{I}^P \wedge \\ & C = L : \text{stop}; \end{cases}$$

The most concrete preserved property is

$$\delta_{\text{t}^{\text{opi}}} = \bigsqcup_{P \in \text{IMP}} \{X \subseteq \Sigma^+ \mid \text{Pres}_{P, \overline{\text{t}^{\text{opi}}}}(X)\}$$

where $\text{Pres}_{P, \overline{\text{t}^{\text{opi}}}}(X)$ if and only if:

$$\forall Y \subseteq (\text{P})_+ . Y \subseteq X \Rightarrow \bigcup \{Z \subseteq \Sigma^+ \mid Z = \overline{\text{t}^{\text{opi}}}(Y, \mathcal{I}^P)\} \subseteq X$$

This means that a set of traces X is preserved by opaque predicate insertion if it contains all the traces that can be obtained from traces in X by inserting opaque predicate P^{t} at program points indicated by \mathcal{I}^P .

5.1.3 Dead code elimination

Let $\mathcal{I} : \text{IMP} \rightarrow \wp(\text{Lab})$ be the result of a preliminary static analysis that given a program returns the subset of its labels which correspond to commands that can be eliminated without affecting the behavior of the program (dead code). Usually the preliminary static analysis consists of dead/faint variable analysis. Given a program P , we assume to know the set $\mathcal{I}^P \subseteq \text{lab}[[P]]$ of labels that the preliminary static analysis has classified as dead code. We assume that conditional commands can't be classified as dead code. We write with $\text{lab}(\varsigma)$ the label of the command contained in the state ς , i.e. if $\varsigma = \langle C, \zeta \rangle$ then $\text{lab}(\varsigma) = \text{lab}[[C]]$.

The semantic transformation $\overline{\text{t}^{\text{dce}}} : \wp(\Sigma^+) \times \wp(\text{Lab}) \rightarrow \wp(\Sigma^+)$ is defined as¹⁵:

$$\begin{aligned} \overline{\text{t}^{\text{dce}}}(X, \mathcal{I}^P) &\stackrel{\text{def}}{=} \{\overline{\text{t}^{\text{dce}}}(\sigma, \mathcal{I}^P) \mid \sigma \in X\} \\ \overline{\text{t}^{\text{dce}}}(\sigma, \mathcal{I}^P) &\stackrel{\text{def}}{=} \text{ELIMINATION}(\sigma, \mathcal{I}^P) \end{aligned}$$

The most concrete preserved property is

$$\delta_{\text{t}^{\text{dce}}} = \bigsqcup_{P \in \text{IMP}} \{X \subseteq \Sigma^+ \mid \text{Pres}_{P, \overline{\text{t}^{\text{dce}}}}(X)\}$$

where $\text{Pres}_{P, \overline{\text{t}^{\text{dce}}}}(X)$ if and only if:

$$\forall Y \subseteq (\text{P})_+ . Y \subseteq X \Rightarrow \bigcup \{Z \subseteq \Sigma^+ \mid Z = \overline{\text{t}^{\text{dce}}}(Y, \mathcal{I}^P)\} \subseteq X$$

This means that a set of traces X is preserved by dead code elimination if it contains all the traces that can be obtained from traces in X by eliminating every command indicated by \mathcal{I}^P .

¹⁵See appendix A for the algorithm ELIMINATION

5.1.4 Loop-unrolling

The easiest looping constructs to unroll are for-loops. Whenever a program P includes a for-loop F , we write $F \in \mathbf{fors}(P)$. More formally, $F \in \mathbf{fors}(P)$ iff $F \stackrel{\text{def}}{=} \{G, I\} \cup H$ and $F \subseteq P$. The command $G \stackrel{\text{def}}{=} l^G : X < E \rightarrow \{l^H, l^O\}$, with $l^G \neq h$ and $l^G \neq l^O$, implements a branching named *guard*. As F always starts with the evaluation of its guard, we have that l^G is the entrypoint of F , ${}^{lab}\llbracket F \rrbracket \cap {}^{lab}\llbracket P \setminus F \rrbracket = \emptyset$ and ${}^{suc}\llbracket P \setminus F \rrbracket \cap {}^{lab}\llbracket F \rrbracket \subseteq \{l^G\}$. The guard is satisfied as long as $X \in \text{Var}$ is less¹⁶ than $E \in \text{Exp}$. If the guard is not satisfied, the for-loop ends transferring the control flow at entrypoint $l^O \notin {}^{lab}\llbracket F \rrbracket$. Otherwise, the execution goes on through H , a set of commands named *body*, and eventually through an increment command $I \stackrel{\text{def}}{=} l^I : X := X + E' \rightarrow l^G$, with $l^I \neq l^G$ and $l^I = l^H \vee l^I \in {}^{suc}\llbracket H \rrbracket$ ¹⁷. We formally define H as the collection of all the commands of P that are reachable from G without going through I , i.e., $H \stackrel{\text{def}}{=} \text{lfp}^{\subseteq} \mathbf{flow}(P)$, where $\mathbf{flow}(P)(Q) \stackrel{\text{def}}{=} \{C \in P \setminus \{I\} \mid {}^{lab}\llbracket C \rrbracket = {}^{suc}\llbracket G \rrbracket \vee \exists C' \in Q. {}^{lab}\llbracket C \rrbracket = {}^{suc}\llbracket C' \rrbracket\}$. We require $l^G, l^I \notin {}^{lab}\llbracket H \rrbracket$. We expect both X and the variables in E and E' not to be assigned inside H . We require X not to be used in E or E' . Finite partial trace $\langle G, \zeta \rangle \eta \langle I, \zeta' \rangle \in \langle F \rangle_{\oplus}$ is an iteration of for-loop F , where $\eta \in \langle H \rangle_{\oplus}$; if $H = \emptyset$ then $\eta = \epsilon$. A maximal trace $\sigma \in \langle F \rangle_{\oplus}$ is a sequence of terminating iterations followed by a state with the command at label l^O . Along the trace, the values of E and E' do not change, while the value of X , though constant throughout each iteration, increases by E' from one iteration to another. Thus, if ζ is a context in a state of $\sigma \in \langle F \rangle_{\oplus}$, we can predict how many increments X still has to undergo, i.e., the number of the iterations from ζ till the end of σ ¹⁸. We just need to define $\alpha_F : {}^{con}\llbracket F \rrbracket \rightarrow \mathbb{N}$ such that

$$\alpha_F(\zeta) \stackrel{\text{def}}{=} \begin{cases} \left\lfloor \frac{(\mathcal{E}\llbracket E \rrbracket \zeta - \mathcal{E}\llbracket X \rrbracket \zeta) + (\mathcal{E}\llbracket E' \rrbracket \zeta - \mathcal{E}\llbracket X \rrbracket \zeta)}{\mathcal{E}\llbracket E' \rrbracket \zeta} \right\rfloor & \text{if } \mathcal{E}\llbracket E \rrbracket \zeta \geq \mathcal{E}\llbracket X \rrbracket \zeta \\ 0 & \text{otherwise} \end{cases}$$

We let τ be the total number of iterations of σ . Along $\sigma \in \langle F \rangle_{\oplus}$ iterations are naturally unfolded, i.e., they come sequentially one after another. In the original program F they fold because any command $C \in F$, although occurring in many different iterations, always appears with the same entrypoint ${}^{lab}\llbracket C \rrbracket$.

Loop-unrolling changes labels in the following way: given the so-called unrolling factor $u \in \mathbb{N}$, it makes all and only the occurrences of C at iterations $k \bmod u$ have the same label (with $0 \leq k < \tau$), thus partitioning the iterations

¹⁶For short, we ignore similar kinds of for-loops which use $>$, \leq or \geq as comparison operator

¹⁷Notice that I makes the control flow return to the guard again

¹⁸Actually we only need the environment ρ contained in ζ

of σ into u classes. Only iterations from the same class fold together. So the code of the unrolled loop is u times longer than F and each of its iterations sequentially executes the task of u native iterations. In this work we consider only the case in which the total number of iterations is known (it's constant) and so we can set $u = \tau$. This is also the standard optimizing behavior of most compiler, like gcc. So we assume that $\mathbf{fors}(P)$ contains only the for-loops which have a constant number of iterations¹⁹ and that $\mathbf{iters}(F)$ returns the number of iterations of $F \in \mathbf{fors}(P)$. Let $\mathcal{I} : \mathbf{IMP} \rightarrow \wp(\mathbf{Lab} \times \mathbf{Lab})$ be the result of a preliminary static analysis that given a program returns the for-loops that can be unrolled. It represents the loop by mean a pair of labels which identify the guard and the increment of the loop. Given a program P , we assume to know the set $\mathcal{I}^P \subseteq \mathit{lab}[[P]] \times \mathit{lab}[[P]]$ of pairs of labels that the preliminary static analysis has classified as representative of loops that can be unrolled. We write with $\mathbf{lab}(\zeta)$ the label of the command contained in the state ζ , i.e. if $\zeta = \langle C, \zeta \rangle$ then $\mathbf{lab}(\zeta) = \mathit{lab}[[C]]$. The semantic transformation $\overline{\mathbf{t}^{\text{lu}}} : \wp(\Sigma^+) \times \wp(\mathbf{Lab} \times \mathbf{Lab}) \rightarrow \wp(\Sigma^+)$ is defined as²⁰:

$$\begin{aligned} \overline{\mathbf{t}^{\text{lu}}}(X, \mathcal{I}^P) &\stackrel{\text{def}}{=} \{\overline{\mathbf{t}^{\text{lu}}}(\sigma, \mathcal{I}^P) \mid \sigma \in X\} \\ \overline{\mathbf{t}^{\text{lu}}}(\sigma, \mathcal{I}^P) &\stackrel{\text{def}}{=} \text{UNROLL}(X, \mathcal{I}^P) \end{aligned}$$

The most concrete preserved property is

$$\delta_{\overline{\mathbf{t}^{\text{lu}}}} = \bigsqcup_{P \in \mathbf{IMP}} \{X \subseteq \Sigma^+ \mid \text{Pres}_{P, \overline{\mathbf{t}^{\text{lu}}}}(X)\}$$

where $\text{Pres}_{P, \overline{\mathbf{t}^{\text{lu}}}}(X)$ if and only if:

$$\forall Y \subseteq (\mathbb{P})_+ . Y \subseteq X \Rightarrow \bigcup \{Z \subseteq \Sigma^+ \mid Z = \overline{\mathbf{t}^{\text{lu}}}(Y, \mathcal{I}^P)\} \subseteq X$$

This means that a set of traces X is preserved by loop-unrolling transformation if it contains all the traces that can be obtained from traces in X by substituting the loops at program points indicated by \mathcal{I}^P with their sequences of iterations.

5.1.5 Loop-invariant code motion

In this context we assume that only assignments can be moved outside the loops. Let $\mathcal{I} : \mathbf{IMP} \rightarrow \wp(\mathbf{Lab})$ be the result of a preliminary static analysis that given a program returns the subset of its labels which correspond to commands that can be moved without affecting the behavior of the program. Usually the preliminary static analysis consists of reaching definitions analysis. Given a program P , we assume to know the set $\mathcal{I}^P \subseteq \mathit{lab}[[P]]$

¹⁹We need a preliminary analysis of program P

²⁰See appendix A for the algorithm UNROLL

of labels that the preliminary static analysis has classified as loop invariant. For every $l \in \mathcal{I}^P$ it's trivial to retrieve the for-loop $F \in \mathbf{fors}(P)$ ²¹ which contains the command at label l . This can be also done only observing a trace of the program. So we assume to have a function **entry** that retrieve the label of the loop's guard related to the loop which contains the command with label l , namely $l^G = \mathbf{entry}(l)$ iff $F = \{G, I\} \cup H$, $l \in \mathit{lab}[[H \cup I]]$ and $l^G = \mathit{lab}[[G]]$. We assume also to have a function **exit** that retrieve the label of the command just next the loop which contains the command with label l , namely $l^O = \mathbf{exit}(l)$ iff $F = \{G, I\} \cup H$, $l \in \mathit{lab}[[H \cup I]]$ and $G = l^G : B \rightarrow \{l^H, l^O\}$. Let \hat{L} a co-label of P , i.e. $\hat{L} \notin \mathit{lab}[[P]]$. We write with $\mathbf{lab}(\varsigma)$ the label of the command contained in the state ς , i.e. if $\varsigma = \langle C, \zeta \rangle$ then $\mathbf{lab}(\varsigma) = \mathit{lab}[[C]]$.

The semantic transformation $\overline{\mathfrak{t}^{\text{licm}}} : \wp(\Sigma^+) \times \wp(\text{Lab}) \rightarrow \wp(\Sigma^+)$ is defined as²²:

$$\begin{aligned} \overline{\mathfrak{t}^{\text{licm}}}(X, \mathcal{I}^P) &\stackrel{\text{def}}{=} \{\overline{\mathfrak{t}^{\text{licm}}}(\sigma, \mathcal{I}^P) \mid \sigma \in X\} \\ \overline{\mathfrak{t}^{\text{licm}}}(\sigma, \mathcal{I}^P) &\stackrel{\text{def}}{=} \text{MOTION}(\sigma, \mathcal{I}^P) \end{aligned}$$

The most concrete preserved property is

$$\delta_{\overline{\mathfrak{t}^{\text{licm}}}} = \bigsqcup_{P \in \text{Imp}} \{X \subseteq \Sigma^+ \mid \text{Pres}_{P, \overline{\mathfrak{t}^{\text{licm}}}}(X)\}$$

where $\text{Pres}_{P, \overline{\mathfrak{t}^{\text{licm}}}}(X)$ if and only if:

$$\forall Y \subseteq (\mathcal{I}^P)_+. Y \subseteq X \Rightarrow \bigcup \{Z \subseteq \Sigma^+ \mid Z = \overline{\mathfrak{t}^{\text{licm}}}(Y, \mathcal{I}^P)\} \subseteq X$$

This means that a set of traces X is preserved by loop-invariant code motion transformation if it contains all the traces that can be obtained from traces in X by moving each command, at program points indicated by \mathcal{I}^P , just before (outside) the loop that contains it.

Table 1: Resilience

	\mathfrak{t}^{ef}	$\mathfrak{t}^{\text{opi}}$	$\mathfrak{t}^{\text{dce}}$	\mathfrak{t}^{lu}	$\mathfrak{t}^{\text{licm}}$
Block-reordering	✓	✗	✓	✗	✓
Static graph-based	✓	✗	✓	✗	✓
Dynamic graph-based	✓	✓	✓	✓	✓
Path-based	✗	✗	✓	✗	✓
Abstract const. prop.	✓	✓	✓	✓	✓

²¹Function defined formally in page 47

²²See appendix A for the algorithm MOTION

Results

Let's see an example. Static graph-based watermarking is t^{opi} -ineffective. This means that $\forall s \in \mathcal{S}. \delta_{\text{t}^{\text{opi}}} \not\sqsubseteq \overline{\mathfrak{M}}(s)$. We prove that the negation of this proposition lead to an absurd. So, suppose that $\exists s \in \mathcal{S}. \delta_{\text{t}^{\text{opi}}} \sqsubseteq \overline{\mathfrak{M}}(s)$. Indeed, for every set of traces X we have $\delta_{\text{t}^{\text{opi}}}(X) \subseteq \overline{\mathfrak{M}}(s)(X)$. Take $X = \mathcal{W}_s$. In this case $\overline{\mathfrak{M}}(X) = \mathcal{W}_s$ but clearly $\mathcal{W}_s \not\subseteq \delta_{\text{t}^{\text{opi}}}(\mathcal{W}_s)$. In fact, due to extensivity, $\mathcal{W}_s \subseteq \delta_{\text{t}^{\text{opi}}}(\mathcal{W}_s)$ and by the fact that $\text{Pres}_{P, \overline{\mathfrak{M}}(s)}(\mathcal{W}_s)$ doesn't hold we have that $\mathcal{W}_s \notin \delta_{\text{t}^{\text{opi}}}$. Absurd. The reasoning about resilience of the other techniques is similar.

The formalization of both the watermarking techniques and the distortive attacks in the semantic setting has allowed us to formally prove the resilience of the considered watermarking systems w.r.t. the distortive attacks described above. Table 1 summarizes our results by showing which watermarking systems is resilient (\checkmark) w.r.t. an attack and which one is not (\times). We can observe that path-base watermarking is not resilient w.r.t. edge-flipping but it is resilient w.r.t. dead-code elimination. Interestingly, the watermarking system based on abstract constant propagation is resilient w.r.t. every attack, this means that it embeds the signature in an abstract property that is preserved by all the considered attacks. Thus, if we want to develop a watermarking system resilient to common obfuscation techniques we should encode the signature in an abstract property implied by denotational semantics.

5.2 Secrecy and Transparency

In Section 3.2 we have discussed how different watermarking systems $\mathfrak{A}_1 = \langle \mathfrak{L}_1, \mathfrak{M}_1, \beta_1 \rangle$ and $\mathfrak{A}_2 = \langle \mathfrak{L}_2, \mathfrak{M}_2, \beta_2 \rangle$ could be compared w.r.t. secrecy and transparency based on the comparison of their most powerful harmless attacks $\hat{\rho}_1$ and $\hat{\rho}_2$. Also in this case it may happen that the two abstractions are not comparable. In this case we compare secrecy and transparency w.r.t. particular observations. Let us denote with $\mathfrak{D}^{\text{cf}\mathfrak{g}} \in \text{uco}(\wp(\Sigma^+))$ the abstract interpreter that computes the control flow graph of programs, and with $\mathfrak{D}^{\text{acp}} \in \text{uco}(\wp(\Sigma^+))$ the abstract interpreter that performs the analysis of constant propagation modulo n , with $n \in \mathbb{N}$. Note that $\mathfrak{D}^{\text{cf}\mathfrak{g}}$ is exactly the extraction domain β of block-reordering watermarking 4.4 and $\mathfrak{D}^{\text{acp}}$ is exactly the extraction domain β of abstract constant propagation watermarking 4.3.

Results

Let's see two example. Static graph-based watermarking is not \top -secret w.r.t $\mathfrak{D}^{\text{cf}\mathfrak{g}}$. Let $\rho = \overline{\mathfrak{D}^{\text{cf}\mathfrak{g}}}$, i.e., the semantic counterpart of $\mathfrak{D}^{\text{cf}\mathfrak{g}}$. $\forall P \in \mathcal{P} \forall s, s' \in \mathcal{S}$ we have that $\llbracket \mathfrak{L}(P, \mathfrak{M}(s)) \rrbracket_+^\rho \neq \llbracket \mathfrak{L}(P, \mathfrak{M}(s')) \rrbracket_+^\rho$. In fact ρ is more concrete than the abstraction domain of the technique and so it is able to view differ-

Table 2: Secrecy & Transparency

	Secrecy				Transparency	
	$\mathcal{D}^{\text{cfig}}$		\mathcal{D}^{acp}		$\mathcal{D}^{\text{cfig}}$	\mathcal{D}^{acp}
Block-reordering	-	-	ϕ	\top	\times	\checkmark
Static graph-based	-	-	ϕ	\top	\times	\checkmark
Dynamic graph-based	ϕ	\top	ϕ	\top	\checkmark	\checkmark
Path-based	ϕ	\top	ϕ	\top	\times	\checkmark
Abstract const. prop.	ϕ	\top	-	-	\checkmark	\times

ence between any stegoprogram. Block-reordering watermarking is transparent w.r.t \mathcal{D}^{acp} . Let $\rho = \overline{\mathcal{D}^{\text{acp}}}$, i.e., the semantic counterpart of \mathcal{D}^{acp} . $\forall P \in \mathcal{P} \forall s \in \mathcal{S}$ we have that $(P)_+^\rho = (\mathcal{L}(P, \overline{\mathcal{M}}(s)))_+^\rho$. In fact the reordering of the basick block of the P doesn't alter the values of the program's variables. So the constant propagation, modulo n , computed on P and every possible stegoprogram returns the same results. The reasoning about resilience of the other techniques is similar.

Given the semantic formalization of the considered watermarking systems and of the two observers introduced above, we provide a formal proof of the secrecy and transparency of these watermarking systems w.r.t. $\mathcal{D}^{\text{cfig}}$ and \mathcal{D}^{acp} . Table 2 summarizes our results. For example, block-reordering watermarking is not \top -secret and it is not ϕ -secret, for any possible ϕ , w.r.t. $\mathcal{D}^{\text{cfig}}$. Instead it is \top -secret, and so ϕ -secret for all possible ϕ , w.r.t. an observer which look at \mathcal{D}^{acp} . Moreover, abstract constant propagation watermarking is invisible w.r.t. $\mathcal{D}^{\text{cfig}}$ while it is not invisible w.r.t. \mathcal{D}^{acp} . So, as we can see, all this techniques are not invisible only w.r.t. the observer which are more precise than $\bigcap \{\overline{\mathcal{M}}(s) \mid s \in \mathcal{S}\}$.

6 Conclusions

In this paper we introduce a semantics-based definition of software watermarking that is general enough to allow the specification of the static, abstract and dynamic watermarking techniques. Indeed, all these techniques can be seen as the exploitation of a completeness hole for the insertion of the signature in an efficient way. Only attacks that are complete w.r.t. the semantic encoding of the signature are able to observe the signature and potentially tamper with it. This means that the abstract domain used for the semantic encoding of the signature $\overline{\mathcal{M}}(s)$ acts like a secret key that allows to disclose the signature to attackers that are complete w.r.t. $\overline{\mathcal{M}}(s)$.

Regarding the quality of a watermarking scheme our general framework provides a formal setting where to prove the efficiency of a watermarking

scheme w.r.t. resilience, secrecy, transparency and accuracy. To validate our theory we have proved the efficiency of five known watermarking systems. Thus, we provide a general theory where researchers can provide formal evidence of quality of the watermarking system that they propose. We believe that this is an important contribution that can be considered as the first step towards a formal theory for software-watermarking where new and existing techniques can be certified w.r.t. their efficiency.

References

- [1] C. COLLBERG, C. D. THOMBORSON, and D. LOW. “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”. In: *Proc. of Conf. Record of the 25th ACM Symp. on Principles of Programming Languages (POPL '98)*. ACM Press, 1998, pp. 184–196.
- [2] C. S. COLLBERG and C. D. THOMBORSON. “Software Watermarking: Models and Dynamic Embeddings.” In: *POPL*. Ed. by A. W. APPEL and A. AIKEN. ACM, 1999, pp. 311–324. ISBN: 1-58113-095-3.
- [3] C. S. COLLBERG and C. D. THOMBORSON. “Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection.” In: *IEEE Trans. Software Eng.* 28.8 (2002), pp. 735–746.
- [4] C. S. COLLBERG, C. D. THOMBORSON, and D. LOW. *A Taxonomy of Obfuscating Transformations*. Tech. rep. 148. Department of Computer Sciences, The University of Auckland, 1997.
- [5] C. S. COLLBERG et al. “Dynamic path-based software watermarking”. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pp. 107–118.
- [6] 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)*. ACM Press, 1977, pp. 238–252.
- [7] 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)*. ACM Press, 1979, pp. 269–282.
- [8] P. COUSOT and R. COUSOT. “An abstract interpretation-based framework for software watermarking”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Venice, Italy, January 14-16, 2004*, pp. 173–185.

- [9] P. COUSOT and R. COUSOT. “Systematic Design of Program Transformation Frameworks by Abstract Interpretation”. In: *Conference Record of the Twenty-ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon: ACM Press, New York, NY, 2002, pp. 178–190.
- [10] M. DALLA PREDÀ and R. GIACOBÀZZI. “Control Code Obfuscation by Abstract Interpretation”. In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 7-9 September, Koblenz, Germany. 2005, pp. 301–310.
- [11] M. DALLA PREDÀ, I. MASTROENI, and R. GIACOBÀZZI. “A Formal Framework for Property-Driven Obfuscation Strategies”. In: *Fundamentals of Computation Theory - 19th International Symposium, FCT 2013, Liverpool, UK, August 19-21, 2013. Proceedings*. 2013, pp. 133–144.
- [12] R. L. DAVIDSON and N. MYHRVOLD. *Method and System for Generating and Auditing a Signature for a Computer Program*. US Patent number 5,559,884. 1996.
- [13] R. GIACOBÀZZI and E. QUINTARELLI. “Incompleteness, counterexamples and refinements in abstract model-checking”. In: *Proc. of The 8th Internat. Static Analysis Symp. (SAS’01)*. Ed. by P. COUSOT. Vol. 2126. Lecture Notes in Computer Science. Springer-Verlag, 2001, pp. 356–373.
- [14] R. GIACOBÀZZI, F. RANZATO, and F. SCOZZARI. “Making Abstract Interpretation Complete”. In: *Journal of the ACM* 47.2 (2000), pp. 361–416.
- [15] R. GIACOBÀZZI. “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)*. IEEE Press., 2008, pp. 7–20.
- [16] R. GIACOBÀZZI and I. MASTROENI. “Abstract Non-interference: Parameterizing Non-interference by Abstract Interpretation”. In: *SIGPLAN Not.* 39.1 (2004), pp. 186–197. ISSN: 0362-1340.
- [17] R. GIACOBÀZZI and I. MASTROENI. “Compositionality in the puzzle of semantics”. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’02)*, Portland, Oregon, USA, January 14-15. 2002, pp. 87–97.
- [18] R. GIACOBÀZZI and I. MASTROENI. “Transforming Abstract Interpretations by Abstract Interpretation.” In: *SAS*. Ed. by M. ALPUENTE and G. VIDAL. Vol. 5079. Lecture Notes in Computer Science. Springer, Aug. 12, 2008, pp. 1–17. ISBN: 978-3-540-69163-1.
- [19] I. MASTROENI. “On the Rôle of Abstract Non-interference in Language-Based Security”. In: *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, Proceedings*. 2005, pp. 418–433.

- [20] S. A. MOSKOWITZ and M. COOPERMAN. *Method for stega-cipher protection of computer code*. US patent 5.745.569. Assignee: The Dice Company, 1996.
- [21] J. NAGRA, C. D. THOMBORSON, and C. S. COLLBERG. "A Functional Taxonomy for Software Watermarking". In: *Aust. Comput. Sci. Commun.* 24.1 (2002), pp. 177–186.
- [22] R. VENKATESAN, V. V. VAZIRANI, and S. SINHA. "A Graph Theoretic Approach to Software Watermarking." In: *Information Hiding*. Ed. by I. S. MOSKOWITZ. Vol. 2137. Lecture Notes in Computer Science. Springer, 2001, pp. 157–168. ISBN: 3-540-42733-3.

Appendices

A Algorithms

Algorithm 1 ELIMINATION

Input σ, \mathcal{I}

```
1: for all  $l \in \mathcal{I}$  do
2:   Let  $\sigma_i$  such that  $\text{lab}(\sigma_i) = l$ 
3:    $l' \leftarrow \text{lab}(\sigma_{i+1})$ 
4:    $j \leftarrow 0$ 
5:   while  $j < |\sigma|$  do                                     //  $|\sigma|$  is updated at every cycle
6:     Let  $\sigma_j = \langle C_j, \zeta_j \rangle$ 
7:     if  $\text{lab}(\sigma_j) = l$  then
8:       Let  $\sigma_{j+1} = \langle C_{j+1}, \zeta_{j+1} \rangle$ 
9:        $\sigma_{j+1} \leftarrow \langle C_{j+1}, \zeta_j \rangle$ 
10:       $\sigma_j \leftarrow \emptyset$ 
11:       $j \leftarrow j - 1$ 
12:     else
13:       if  $C_j = L : A \rightarrow l$ ; then
14:          $\sigma_j \leftarrow \langle L : A \rightarrow l', \zeta_j \rangle$ 
15:       if  $C_j = L : B \rightarrow \{l, L_F\}$ ; then
16:          $\sigma_j \leftarrow \langle L : B \rightarrow \{l', L_F\}, \zeta_j \rangle$ 
17:       if  $C_j = L : B \rightarrow \{L_T, l\}$ ; then
18:          $\sigma_j \leftarrow \langle L : B \rightarrow \{L_T, l'\}, \zeta_j \rangle$ 
19:        $j \leftarrow j + 1$ 
```

Output σ

Algorithm 2 UNROLL

- Every label $l \in {}^{lab}[P]$ mustn't have any number for apex

Input σ, \mathcal{I}

```

1: for all  $\langle l^G, l^I \rangle \in \mathcal{I}$  do
2:   Let  $\sigma_i = \langle C_i, \zeta_i \rangle$  such that  $\mathbf{lab}(\sigma_i) = l^G$ 
3:   Let  $\sigma_j = \langle C_j, \zeta_j \rangle$  such that  $\mathbf{lab}(\sigma_j) = l^I$ 
4:   Let  $X$  be the variable of the guard in  $C_i$ , i.e.  $C_i = l^G : X < E \rightarrow \{l^H, l^O\}$ ;
5:   Let  $X$  be the variable in the increment  $C_j$ , i.e.  $C_j = l^I : X := X + \dot{E} \rightarrow l^G$ ;
6:   Let  $\dot{e}$  be the value of expression  $\dot{E}$  computed in context  $\zeta_j$ 
7:    $\mathit{iters} \leftarrow$  ordered list of pairs  $\langle i, j \rangle$  such that:
       $i < j, \mathbf{lab}(\sigma_i) = l^G, \mathbf{lab}(\sigma_j) = l^O, \forall k \in (i, j). \mathbf{lab}(\sigma_k) \neq l^O$ 
8:   for all  $\langle i, j \rangle \in \mathit{iters}$  do // in list order
9:      $m \leftarrow 0$ 
10:    for  $k = i$  to  $j - 1$  do
11:      Let  $\sigma_k = \langle C_k, \langle \rho_k, \iota_k \rangle \rangle$ 
12:      if  $\mathbf{lab}(\sigma_k) = l^G$  then
13:         $m \leftarrow m + 1$ 
14:        if  $m > 1$  then
15:           $L \leftarrow l^{G^{m-1}}$ 
16:           $\rho_k \leftarrow \rho_k[X \leftarrow \rho_k(X) - m\dot{e}]$ 
17:        else
18:           $L \leftarrow l^G$ 
19:           $\sigma_k \leftarrow \langle L : \mathbf{skip} \rightarrow l^{H^m}; \langle \rho_k, \iota_k \rangle \rangle$ 
20:        if  $\mathbf{lab}(\sigma_k) = l^I$  then
21:           $\sigma_k \leftarrow \langle l^{I^m} : \mathbf{skip} \rightarrow l^{G^m}; \langle \rho_k[X \leftarrow \rho_k(X) - m\dot{e}], \iota_k \rangle \rangle$ 
22:        if  $\mathbf{lab}(\sigma_k) \neq l^G \wedge \mathbf{lab}(\sigma_k) \neq l^I$  then // so  $C_k \in H$ 
23:          if  $C_k = L : B \rightarrow \{L_T, L_F\}$ ; then
24:             $B' \leftarrow B[X \leftarrow X + (m - 1)\dot{e}]$ 
25:             $C_k \leftarrow L^m : B' \rightarrow \{L_T^m, L_F^m\}$ ;
26:          if  $C_k = L_1 : A \rightarrow L_2$ ; then
27:             $A' \leftarrow A[X \leftarrow X + (m - 1)\dot{e}]$ 
28:             $C_k \leftarrow L_1^m : A' \rightarrow L_2^m$ ;
29:           $\sigma_k \leftarrow \langle C_k, \langle \rho_k[X \leftarrow \rho_k(X) - m\dot{e}], \iota_k \rangle \rangle$ 
30:    Let  $\sigma_{j-1} = \langle C_{j-1}, \zeta_{j-1} \rangle$ 
31:     $\sigma \leftarrow \sigma_0 \dots \sigma_i \dots \sigma_{j-1} \langle l^{H^m} : X := X + (m - 1)\dot{e} \rightarrow l^O; \zeta_{j-1} \rangle \sigma_j \dots \sigma_{|\sigma|-1}$ 

```

Output σ

Algorithm 3 MOTION

Input σ, \mathcal{I}

```
1: for all  $l \in \mathcal{I}$  do
2:   Let  $\sigma_k = \langle C_l, \zeta_k \rangle$  such that  $\mathbf{lab}(\sigma_k) = l$ 
3:   Let  $X$  be the variable assigned in  $C_l$ , i.e.  $C_l = l : X := E_l \rightarrow l'$ ;
4:   Let  $e$  be the value of expression  $E_l$  computed in context  $\zeta_j$ 
5:    $l^G \leftarrow \mathbf{entry}(l)$ 
6:    $l^O \leftarrow \mathbf{exit}(l)$ 
7:    $\mathit{iters} \leftarrow$  ordered list of pairs  $\langle i, j \rangle$  such that:
       $i < j, \mathbf{lab}(\sigma_i) = l^G, \mathbf{lab}(\sigma_j) = l^O, \forall k \in (i, j). \mathbf{lab}(\sigma_k) \neq l^O$ 
8:   Let  $\langle i, j \rangle$  the first element of  $\mathit{iters}$ 
9:    $w \leftarrow 0$ 
10:  for all  $\langle i, j \rangle \in \mathit{iters}$  do // in list order
11:     $\sigma' \leftarrow \sigma_w \dots \sigma_{i-2}$ 
12:    Let  $\sigma_{i-1} = \langle C_{i-1}, \zeta_{i-1} \rangle$ 
13:    if  $C_{i-1} = L : A \rightarrow l^G$ ; then
14:       $\sigma' \leftarrow \sigma' \langle L : A \rightarrow \hat{L}; \zeta_{i-1} \rangle \langle \hat{L} : X := E_l \rightarrow l^G, \zeta_{i-1} \rangle$ 
15:    if  $C_{i-1} = L : B \rightarrow \{l^G, L_F\}$ ; then
16:       $\sigma' \leftarrow \sigma' \langle L : B \rightarrow \{\hat{L}, L_F\}; \zeta_{i-1} \rangle \langle \hat{L} : X := E_l \rightarrow l^G, \zeta_{i-1} \rangle$ 
17:    if  $C_{i-1} = L : B \rightarrow \{L_T, l^G\}$ ; then
18:       $\sigma' \leftarrow \sigma' \langle L : B \rightarrow \{L_T, \hat{L}\}; \zeta_{i-1} \rangle \langle \hat{L} : X := E_l \rightarrow l^G, \zeta_{i-1} \rangle$ 
19:    for all  $k \in [i, j]$  do
20:      Let  $\sigma_k = \langle C_k, \langle \rho_k, \iota_k \rangle \rangle$ 
21:       $\sigma_k \leftarrow \langle C_k, \langle \rho_k[X \leftarrow e], \iota_k \rangle \rangle$ 
22:      if  $\mathbf{lab}(\sigma_k) \neq l$  then
23:        if  $C_k = L : A \rightarrow l$  then
24:           $\sigma_k \leftarrow \langle L : A \rightarrow l'; \langle \rho_k, \iota_k \rangle \rangle$ 
25:        if  $C_k = L : B \rightarrow \{l, L_F\}$ ; then
26:           $\sigma_k \leftarrow \langle L : B \rightarrow \{l', L_F\}; \langle \rho_k, \iota_k \rangle \rangle$ 
27:        if  $C_k = L : B \rightarrow \{L_T, l\}$ ; then
28:           $\sigma_k \leftarrow \langle L : B \rightarrow \{L_T, l'\}; \langle \rho_k, \iota_k \rangle \rangle$ 
29:         $\sigma' \leftarrow \sigma' \sigma_k$ 
30:     $w \leftarrow j$ 
31:    Let  $\langle i, j \rangle$  the last element of  $\mathit{iters}$ 
32:     $\sigma' \leftarrow \sigma' \sigma_j \dots \sigma_{|\sigma|-1}$ 
33:     $\sigma \leftarrow \sigma'$ 
34:   $i \leftarrow 0$ 
35:  for  $j = 0$  to  $|\sigma| - 1$  do
36:    if  $\mathbf{lab}(\sigma_j) \notin \mathcal{I}$  then
37:       $\sigma'_i \leftarrow \sigma_j$ 
38:       $i \leftarrow i + 1$ 
Output  $\sigma'$ 
```
