Under consideration for publication in Formal Aspects of Computing

Maximal incompleteness as obfuscation potency

Roberto Giacobazzi[1] and Isabella Mastroeni and Mila Dalla Preda

Università degli Studi di Verona, Verona, Italy
and [1] IMDEA Software Institute, Madrid, Spain

Abstract.

Obfuscation is the art of making code hard to reverse engineer and understand. In this paper, we propose a formal model for specifying and understanding the strength of obfuscating transformations with respect to a given attack model. The idea is to consider the attacker as an abstract interpreter willing to extract information about the program's semantics. In this scenario, we show that obfuscating code is making the analysis imprecise, namely making the corresponding abstract domain incomplete. It is known that completeness is a property of the abstract domain and the program to analyse. We introduce a framework for transforming abstract domains, i.e., analyses, towards incompleteness. The family of incomplete abstractions for a given program provides a characterisation of the potency of obfuscation employed in that program, i.e., its strength against the attack specified by those abstractions. We show this characterisation for known obfuscating transformations used to inhibit program slicing and automated disassembly.

Keywords: Abstract interpretation, static program analysis, program semantics, program transformation, lattice theory, closure operators, code obfuscation.

1. Introduction

Obfuscation is the production of misleading, ambiguous and plausible but confusing information as an act of concealment or evasion [BN15]. In this scenario, code obfuscation is the art of making programs hard to understand and to reverse engineer with the purpose of concealing information such as cryptographic keys or critical data/control structures [CN09]. An obfuscator is a semantics-preserving transformation that, while preserving the input/output program relation, makes its internal structure and behavior extremely hard to analyse. While a corpus of results has been obtained on the theoretical foundation of code and circuit obfuscation as cryptographic transformations (e.g., see the impossibility result in [BGI$^+$12] and recent achievements on indistinguishability obfuscation in [GGH$^+$13]), little is known from the perspective of programming languages.

Attacking code means interpreting code, with the intention of extracting and exploiting its extensional properties from their intentional representation. The idea is that, because of time/space and effectiveness constraints, when dealing with complex programs, the attack is necessarily approximated. An attack is

Correspondence and offprint requests to: Prof. Roberto Giacobazzi, University of Verona and IMDEA Software Institute. Email: roberto.giacobazzi@univr.it therefore the combination of a number of automatic and semi-automatic tools (e.g., see the IDA Pro suite of tools) such as: static and dynamic analyses, SAT solvers, theorem provers, program monitors, disassemblers, decompilers, program slicers, code profilers and tracers, emulators etc. All these tools involve approximate automatic interpretation. Any attack is therefore inherently based on approximated interpretation. In this context, protecting programs from an attacker means making the approximate interpretation, on which the attacker is based, harmless.

This idea has been studied in [DG09, JGM12, GM12]. As an intuitive example, if the attacker is a finite state automaton, it is always possible to transform any program P into an equivalent program P' which is obscure for such an attacker. This can be obviously obtained by embedding a push-down automaton in P computing data beyond the size of the finite state automata modelling the attacker. The Pumping Lemma can therefore be used as a simple example of what such an attacker cannot observe about the transformed (obscured) program. By following this simple argument, the notion of *approximate obfuscation* has been introduced in [Gia08], and it has been specified mathematically in [JGM12] in terms of abstract interpretation [CC77, CC79b].

An attacker is an abstract interpretations of the program on an abstract domain of approximate data. It is known (see [CC79b]) that the precision of an abstract interpretation is determined by the chosen abstract domain. For this reason we identify attackers with abstract domains. In this scenario, obfuscating a program with respect to an attacker specified as an approximate (abstract) interpreter means making this interpreter imprecise [Gia08]. This is precisely modelled by the notion of incompleteness of an abstract interpretation [GRS00]. An abstract interpretation is complete for a program P if no error is made by the abstract interpreter with respect to the abstraction of the concrete interpretation. Given a program P and an abstract domain $\mathcal{A}$ we use $[\![P]\!]^{\mathcal{A}}$ to denote the abstract interpretation of P on the abstract domain $\mathcal{A}$, i.e., its abstract computation, and we use $\mathcal{A}([\![P]\!])$ to denote the abstraction of the concrete semantics of P on the abstract domain $\mathcal{A}$. We have completeness when $\mathcal{A}([\![P]\!]) = [\![P]\!]^{\mathcal{A}}$. Consider, for instance, the simple program P : $\mathtt{x} = \mathtt{a} * \mathtt{b}$, multiplying $\mathtt{a}$ and $\mathtt{b}$, and storing the result in $\mathtt{x}$. An attacker observing the sign abstraction $\mathcal{A} = \{\top, -, 0, +, \bot\}$ is able to catch, with no loss of precision, the intended sign behavior of P because the sign abstraction $\mathcal{A}$ is precise for integer multiplication. If we replace P with $\mathfrak{O}(P)$: $\mathtt{x} = 1$; $\mathtt{if}\ \mathtt{b} \leq 0\ \mathtt{then}\ \{\mathtt{a} = -\mathtt{a}; \mathtt{b} = -\mathtt{b}\}$; $\mathtt{while}\ \mathtt{b} \neq 0\ \{\mathtt{x} = \mathtt{a} + \mathtt{x}; \mathtt{b} = \mathtt{b} - 1\}$; $\mathtt{x} = \mathtt{x} - 1$; we obfuscate the observer $\mathcal{A}$ because the rule of signs is incomplete for integer addition, in particular we observe that when $a = -$ and $b = +$, then $x = +$ (being $\mathcal{A}(1) = +$) and $a + x$ is the sum of a negative number with a positive one, and therefore we lose the sign information.

In this paper, we are interested in modelling the potency of an obfuscation. This means that, given a program P, we characterise the family of attackers for which P is maximally obscure, i.e., such that the abstract interpretation of P is maximally incomplete. These attackers represent the potency of the obfuscation employed in P. We introduce abstract domain transformers that maximize the incompleteness of the corresponding abstract interpreter. This means that, if $\mathcal{A}$ is an abstract domain and $[\![P]\!]^{\mathcal{A}}$ is a complete abstract interpretation of P on the abstract domain $\mathcal{A}$, i.e., $\mathcal{A}([\![P]\!]) = [\![P]\!]^{\mathcal{A}}$, then it is possible to transform $\mathcal{A}$ into an abstract domain $\mathcal{I}(\mathcal{A})$ such that $[\![P]\!]^{\mathcal{I}(\mathcal{A})}$ is maximally incomplete, and therefore imprecise. The process of making an abstraction incomplete is indeed the inverse of the completeness refinement. This is achieved by considering the adjoint operations of the completeness refinement transformations introduced in [GRS00]. In particular, we introduce the notion of *incomplete compressor* which removes, from a given abstract domain $\mathcal{A}$, those elements that are useful for improving the precision of an abstract interpretation on $\mathcal{A}$. Although clearly contradictory with respect to the common practice in program analysis, which is achieving precision, making abstractions incomplete provides an unexpectedly useful model for understanding code obfuscation, in particular when modelling the potency or strength of an obfuscation. In this setting, we prove that, the more an attacker $\mathcal{A}$ is close to the maximally incomplete domain $\mathcal{I}(\mathcal{A})$ for a given obfuscated code $\mathfrak{O}(P)$, the more $\mathfrak{O}$ is a successful code obfuscation.

Interestingly, the abstract domain compression that computes the maximally incomplete domain $\mathcal{I}(\mathcal{A})$ associated with an attacker $\mathcal{A}$ suggests the design of $\mathfrak{O}$, i.e., it identifies precisely the information on which the obfuscation $\mathfrak{O}$ has to act in order to successfully obfuscate P with respect to $\mathcal{A}$. We apply this idea to known obfuscation of code used to prevent known techniques for program analysis and understanding, such as program slicing and disassembly. In this setting we are able to prove that what these obfuscating transformations do is precisely inducing the maximal degree of incompleteness in the analysis of the obfuscated code. This paper makes the following contributions:

- The definition of domain transformers that make a domain maximally incomplete for the analysis of a given program (extended and revised version of [GM12])
- The characterisation of the potency of code obfuscation in terms of incomplete abstract domains as obtained by our transformers:
  - We provide a formal characterisation for the potency of an obfuscating transformer with respect to a program semantics and to an abstract analysis (the attacker) of the considered semantics.
  - We validate the proposed characterisation by formally proving the potency of known obfuscating transformations in inhibiting program slicing and automatic disassembly.
  - We discuss how our framework can be used for building, in a semi-automatic way, an obfuscator able to defeat a given attacker. This point relies on the notion of code obfuscation as partial evaluation of distorted interpreters introduced un [JGM12].

The paper is structured as follows. In Section 2, we give the basic mathematical notation, a brief introduction to abstract interpretation, adjoint closure operators, soundness and completeness and semantics for a simple imperative language. In Section 3, we introduce the incomplete compression of an abstract domain and prove its basic algebraic properties. We also prove that incompleteness cannot be systematically derived by expanding abstract domains. In Section 4, we apply incomplete compressors to formalize a characterisation of the potency range of a given obfuscation and then we use this framework for measuring the strength of known obfuscations preventing program slicing and disassembling. In Section 5 we discuss how the theoretical investigation of this paper can be used to provide insights on how to build an obfuscator that defeats a given attacker.

2. Preliminaries

We consider the standard abstract domain definition as formalized in [CC77] and [CC79b] in terms of Galois connections. It is well known that this is a restriction for abstract interpretation because relevant abstractions do not form Galois connections and Galois connections are not expressive enough for modelling dynamic fix-point approximation, e.g., by fix-point widening [CC92b]. In this section we introduce the basic mathematical background concerning Galois-connection based abstract interpretation, residuated closures, fix-point soundness and completeness.

2.1. Basic lattice and fix-point theory

If S and T are sets, then $\wp(S)$ denotes the powerset of S, $|S|$ the cardinality of S, $S \smallsetminus T$ the set-difference between S and T, $S \subset T$ strict inclusion, $S \times T$ the cartesian product, and for a function $f : S \to T$ and $X \subseteq S$, $f(X) \stackrel{\text{def}}{=} \{f(x) \mid x \in X\}$. By $g \circ f$ we denote the composition of the functions f and g, i.e., $g \circ f \stackrel{\text{def}}{=} \lambda x.g(f(x))$. In the sake of simplicity, in the following we will omit parentheses when composing functions. $\langle P, \leq\rangle$ denotes a poset P with ordering relation $\leq$, while $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$ denotes a complete lattice on the set C, with ordering $\leq$, least upper bound (*lub*) $\vee$, greatest lower bound (*glb*) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. In the following, we will often abuse notation by denoting as C the complete lattice. Often, $\leq_P$ will be used to denote the underlying ordering of a poset P, and $\vee_C$, $\wedge_C$, $\top_C$ and $\bot_C$ to denote the basic operations and elements of a complete lattice C. The notation $C \cong D$ denotes that C and D are isomorphic ordered structures. Let P be a poset and $S \subseteq P$. Then, $\max(S) \stackrel{\text{def}}{=} \{x \in S \mid \forall y \in S.\ x \leq_P y \Rightarrow x = y\}$ denotes the set of maximal elements of S in P; also, the downward closure of S is defined by $\downarrow S \stackrel{\text{def}}{=} \{x \in P \mid \exists y \in S.\ x \leq_P y\}$, and for $x \in P$, $\downarrow x$ is a shorthand for $\downarrow\{x\}$, while the upward closure $\uparrow$ is dually defined. In a complete lattice $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$ an element $x \in C$ is *meet-irreducible* if $x \neq \top$ and if $x = y \wedge z$ then $x = y$ or $x = z$. Namely, x is meet-irreducible if it cannot be obtained as glb of two other elements. We denote with $\mathit{Mirr}(C)$ the set of meet-irreducible elements of lattice C.

We use the symbol $\sqsubseteq$ to denote pointwise ordering between functions: If S is any set, P a poset, and $f, g : S \to P$ then $f \sqsubseteq g$ if for all $x \in S$, $f(x) \leq_P g(x)$. An operator $f : P \longrightarrow P$ is extensive if $\forall x \in P.\ x \leq_P f(x)$. It is reductive if $\forall x \in P.\ x \geq_P f(x)$. Let C and D be complete lattices. Then, $C \xrightarrow{\text{m}} D$ and $C \xrightarrow{\text{c}} D$ denote, respectively, the set and the type of all monotone and (Scott-)continuous functions from C to D. Recall that $f \in C \xrightarrow{c} D$ if and only if f preserves *lub*'s of (nonempty) chains if and only if f preserves *lub*'s of directed subsets. Also, $f : C \to D$ is (completely) additive if f preserves *lub*'s of all subsets of C (emptyset included), while co-additivity is dually defined. The *additive lift* of $f : C \to D$ is a function $f^a : \wp(C) \to \wp(D)$ such that $f^a \overset{\text{def}}{=} \lambda X.\ \{\ f(x) \mid\ x \in X\ \ \}$.

lfp(f) and *gfp*(f) denote respectively the least and greatest fix-point, when they exist, of an operator f on a poset. The well-known Knaster-Tarski's theorem states that any monotone operator $f : C \xrightarrow{m} C$ on a complete lattice C admits both least and greatest fix-points, and the following characterisations hold:

$$\mathit{lfp}(f) = \bigwedge_C \{x \in C \mid f(x) \leq_C x\} \;\text{ and }\; \mathit{gfp}(f) = \bigvee_C \{x \in C \mid x \leq_C f(x)\}.$$

Let us note that if $f, g : C \xrightarrow{m} C$ and $f \sqsubseteq g$ then $\mathit{lfp}(f) \sqsubseteq \mathit{lfp}(g)$. It is known that if $f : C \xrightarrow{c} C$ is continuous then $\mathit{lfp}(f) = \vee_{i \in \mathbb{N}} f^i(\bot_C)$, where, for any $i \in \mathbb{N}$ and $x \in C$, the i-th power of f in x is inductively defined as follows: $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$. Dually, if $f : C \to C$ is co-continuous then $\mathit{gfp}(f) = \wedge_{i \in \mathbb{N}} f^i(\top_C)$. $\{f^i(\bot_C)\}_{i \in \mathbb{N}}$ and $\{f^i(\top_C)\}_{i \in \mathbb{N}}$ are called, respectively, the upper and lower Kleene's iteration sequences of f. The set of all finite sequences (traces) over an alphabet Σ is denoted Σ^+. If $\sigma, \sigma' \in \Sigma^+$ then $\sigma\sigma' \in \Sigma^+$ is the concatenation of the two sequences.

2.2. Abstract domains individually and collectively

Concrete domains represent collections of computational objects on which the concrete semantics and models are defined. These include standard data-types (e.g., heap, stack, numerical types), control-flow structures, etc. Abstract domains are collections of approximate objects, representing properties of concrete objects in a domain-like structure. The relation between concrete and abstract domains can be specified in terms of *Galois connections*, and this sets up the so called standard adjoint framework of abstract interpretation [CC77]. The adjoint presentation is a relatively restrictive view of abstract interpretation. Weaker frameworks could involve the weakening of the relation between concrete and abstract domains, e.g. in [CC92a], or sophisticated fix-point iteration strategies by *fix-point widening* on approximate domains [CC92b]. In this paper we consider abstractions in the standard adjoint framework, which provides the richest mathematical environment for proving properties about abstractions. More formally, if $\langle C, \leq, \top, \bot, \vee, \wedge \rangle$ is a complete lattice, a pair of monotone functions $\alpha : C \xrightarrow{m} A$ and $\gamma : A \xrightarrow{m} C$ forms an *adjunction* or a *Galois connection* if for any $x \in C$ and $y \in A$: $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. α [resp. γ] is the *left-* [*right-*]*adjoint* to γ [α] and it is additive [co-additive], i.e., it preserves *lub*'s [*glb*] of all subsets of the domain (emptyset included). Let us recall that the right adjoint of a function f, when it exists, is defined as $f^+ \overset{\text{def}}{=} \lambda x.\ \bigvee\{\ y \mid\ f(y) \leq x\ \ \}$. Conversely the left adjoint of a function f, when it exists, is defined as $f^- \overset{\text{def}}{=} \lambda x.\ \bigwedge\{\ y \mid\ x \leq f(y)\ \ \}$. In Galois connections $\gamma^- = \alpha$ and $\alpha^+ = \gamma$. Abstract domains can be also equivalently formalized as closure operators on the concrete domain [CC79b]. An *upper [lower] closure operator* $\rho : P \longrightarrow P$ on a poset P is monotone, idempotent, and extensive [reductive]. Closures are uniquely determined by their fix-points $\rho(C)$. In the following, we will often use closures both as functions and as sets (viz., domains). Given $X \subseteq C$, the least abstract domain containing X is the least closure including X as fix-points, which is the *Moore-closure* or *Moore family* of X defined as: $\mathcal{M}(X) \overset{\text{def}}{=} \{\bigwedge S \mid S \subseteq X\}$. Dual Moore-closures and families are defined by duality. It turns out that $\langle \rho(C), \leq \rangle$ is a complete meet subsemilattice of C (i.e., $\wedge$ is its *glb*), but, in general, it is not a complete sublattice of C, since the *lub* in $\rho(C)$ — defined by $\lambda Y \subseteq \rho(C).\, \rho(\vee Y)$ — might be different from that in C. In fact, $\rho(C)$ is a complete sublattice of C if and only if ρ is additive. The set of all upper [lower] closure operators on P is denoted by *uco*(P) [*lco*(P)]. The *lattice of abstract domains* of C, is isomorphic to *uco*(C), (cf. [CC77, Section 7] and [CC79b, Section 8]). Recall that if C is a complete lattice, then $\langle \mathit{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \mathit{id} \rangle$ is a complete lattice, where $\mathit{id} \overset{\text{def}}{=} \lambda x.x$ and for every $\rho, \eta \in \mathit{uco}(C)$, $x \in C$ and $\{\rho_i\}_{i \in I} \subseteq \mathit{uco}(C)$ (where I is a set of indexes that identify a subset of the closure operators of *uco*(C)) we have that: $\rho \sqsubseteq \eta$ if and only if $\forall y \in C.\ \rho(y) \leq \eta(y)$ if and only if $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i \in I} \rho_i)(x) = \wedge_{i \in I} \rho_i(x)$; $(\sqcup_{i \in I} \rho_i)(x) = x \Leftrightarrow \forall i \in I.\ \rho_i(x) = x$; $\lambda x.\top$ is the top element and $\lambda x.x$ is the bottom element. Thus, the *glb* in *uco*(C) is defined pointwise, while the *lub* of a set of closures $\{\rho_i\}_{i \in I} \subseteq \mathit{uco}(C)$ is the closure whose set of fix-points is given by the set-intersection

$\sqcap_{i \in I} \rho_i(C)$. In the following, we will make use of the following basic properties for $\rho, \eta \in uco(C)$ and $Y \subseteq C$: $\rho(\wedge \rho(Y)) = \wedge \rho(Y)$; $\rho(\vee Y) = \rho(\vee \rho(Y))$; $\eta \sqsubseteq \rho \Leftrightarrow \eta\rho = \rho \Leftrightarrow \rho \circ \eta = \rho$. Some of the most important operations on upper closure operators are: Reduced product [CFG$^+$95] and pseudo-complement [CFG$^+$95]. The reduced product is the glb operator $\sqcap$ on $uco(C)$ and it is typically used to combine known abstract domains in order to design new abstractions. Pseudo-complement corresponds to the inverse of reduced product, namely an operator that, given two domains $C \sqsubseteq D$, gives as result the most abstract domain $C \ominus D$, whose reduced product with D is exactly C, i.e., $(C \ominus D) \sqcap D = C$. The pseudo-complement of an abstract domain D is defined as: $C \ominus D \overset{\text{def}}{=} \sqcup \{\ E \in uco(C) \mid\ D \sqcap E = C\ \}$. In the adjoint framework of abstract interpretation, A_1 is more precise (viz. more concrete) than A_2 (i.e., A_2 is an abstraction of A_1) if and only if $A_1 \sqsubseteq A_2$ in $uco(C)$ if and only if $A_2 \in uco(A_1)$.

2.3. Adjoining closure operators

In the following we will make an extensive use of adjunction, in particular of closure operators. Janowitz [Jan67] characterised the structure of *residuated* (adjoint on a complete lattice) closure operators by the following basic result (see also [BJ72]).

Theorem 2.1. [Jan67, Theorem 2.10]
Let $f : C \longrightarrow C$ be a *residuated* map, i.e., $\langle f, f^+ \rangle$ is a pair of adjoint operators on the complete lattice C, then

$$(1)\ \ f \in uco(C) \ \Leftrightarrow\ f^+ \in lco(C) \ \Leftrightarrow\ f \circ f^+ = f^+ \ \Leftrightarrow\ f^+ \circ f = f$$

and

$$(2)\ \ f \in lco(C) \ \Leftrightarrow\ f^+ \in uco(C) \ \Leftrightarrow\ f \circ f^+ = f \ \Leftrightarrow\ f^+ \circ f = f^+$$

Dually, if $f : C \longrightarrow C$ is a *dual-residuated*[1] map and f^- is its left-adjoint (defined in the previous section), then [MG15]

$$(3)\ \ f \in uco(C) \ \Leftrightarrow\ f^- \in lco(C) \ \Leftrightarrow\ f \circ f^- = f \ \Leftrightarrow\ f^- \circ f = f^-$$

and

$$(4)\ \ f \in lco(C) \ \Leftrightarrow\ f^- \in uco(C) \ \Leftrightarrow\ f^- \circ f = f \ \Leftrightarrow\ f \circ f^- = f^-$$

Let $\tau \in lco(C)$. By Theorem 2.1, if τ^- exists then $\tau^-(\tau(X)) = \tau(X)$ and $\tau(\tau^-(X)) = \tau^-(X)$. This means that τ^- is such that both τ and τ^- have the same sets of fix-points, namely τ^- extends any object X to the largest object Y such that $\tau(Y) = Y$. Conversely, the right adjoint of τ, when it exists, is quite different. By Theorem 2.1, we have that if τ^+ exists then $\tau^+(\tau(X)) = \tau^+(X)$ and $\tau(\tau^+(X)) = \tau(X)$. In this case $\tau^+(X)$ is not a fix-point of τ. Instead, it is the least element Y such that $\tau(X) = X = \tau(Y)$. The following result strengthen Theorem 2.1 by showing the order-theoretic structure of residuated closures.

Proposition 2.2. [MG15] Let $\tau \in lco(C)$ and $\eta \in uco(C)$.

1. If $\langle \tau^-, \tau \rangle$ and $\langle \eta, \eta^+ \rangle$ are pairs of adjoint functions then
$$\tau^- = \lambda X.\ \bigwedge\{\tau(Y) | \tau(Y) \geq X\} \qquad \text{and} \qquad \eta^+ = \lambda X.\ \bigvee\{\eta(Y) | X \geq \eta(Y)\}.$$
2. If $\langle \tau, \tau^+ \rangle$ and $\langle \eta^-, \eta \rangle$ are pairs of adjoint functions then
$$\tau^+ = \lambda X.\ \bigvee\{Y | \tau(Y) = \tau(X)\} \qquad \text{and} \qquad \eta^- = \lambda X.\ \bigwedge\{Y | \eta(X) = \eta(Y)\}.$$

In particular this result leads to the observation that the existence of adjunction is related to the notion of *closure uniformity.* Uniform closures have been introduced in [GR98] for specifying the notion of *abstract domain compression*, namely the operation for reducing abstract domains to their minimal structure with respect to some given abstraction refinement $\eta \in lco(uco(C))$. An upper closure η is *meet-uniform* [GR98] if $\eta(\bigwedge\{Y | \eta(X) = \eta(Y)\}) = \eta(X)$. Join-uniformity is dually defined for lower closures. Well-known non-co-additive upper closures are meet-uniform, such as the downward closure $\downarrow$ of a subset of a partially ordered set [GR98].

[1] Defined by duality.

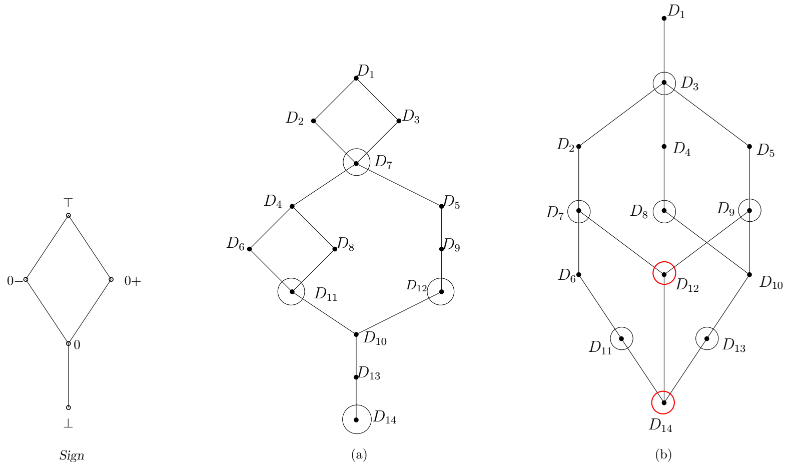


Fig. 1. Lifted *lco*(*Sign*).

It is known that any $\rho \in uco(C)$ is join-uniform and the set of meet-uniform upper closures $uco^*(C)$ is a Moore-family of $uco(C)$. Dually, the same holds for lower closure operators, namely $\tau \in lco(C)$ is meet-uniform and the set of join-uniform lower closures $lco^*(C)$ is a Moore-family of $lco(C)$. As observed in [GR98] when only uniformity holds, the adjoint function may fail monotonicity. In [GR98] the authors proved that the adjoint function is monotone on a lifted order induced by τ. Given a partial order $\leq$, its lifted version is $\leq_\tau \subseteq C \times C$, defined as: $\forall x, y \in C:\ x \leq_\tau y \ \Leftrightarrow\ (\tau(x) \leq \tau(y)) \ \wedge\ (\tau(x) = \tau(y) \ \Rightarrow\ x \leq y)$. $\leq_\tau$ is such that $\leq \Rightarrow \leq_\tau$. The following result is immediate by [Jan67] and Proposition 2.2.

Proposition 2.3. [MG15] Let $\tau \in lco(C)$ $[\eta \in uco(C)]$. $\langle \tau, \tau^+ \rangle$ $[\langle \eta^-, \eta \rangle]$ is a pair of adjoint closures on the lifted order if and only if τ is join-uniform $[\eta$ is meet-uniform$]$.

Example 2.4. Consider the *Sign* domain in Figure 1, *uco*(*Sign*) is the set of all possible abstractions of *Sign*, namely the set of all Moore families over *Sign*, and it is given by the following domains:

$$\begin{array}{cccc}
D_1 = \{\top\} & D_2 = \{\top, 0+\} & D_3 = \{\top, 0\} & D_4 = \{\top, \bot\} \\
D_5 = \{\top, 0-\} & D_6 = \{\top, 0+, \bot\} & D_7 = \{\top, 0+, 0\} & D_8 = \{\top, 0, \bot\} \\
D_9 = \{\top, 0-, 0\} & D_{10} = \{\top, 0-, \bot\} & D_{11} = \{\top, 0+, 0, \bot\} & D_{12} = \{\top, 0+, 0-, 0\} \\
D_{13} = \{\top, 0-, 0, \bot\} & D_{14} = D & &
\end{array}$$

Consider the domain transformer $\tau_a = \lambda X.\, X \sqcap D_7$ that, given an abstract domain in *uco*(*Sign*) computes its glb with the domain D_7, namely it returns the most abstract domain that expresses the information of both the input domain X and domain D_7. The lco domain with respect to the lifted order $\sqsubseteq_{\tau_a}$ of the uco standard order $\sqsubseteq$, is depicted in Fig 1(a), where the circled domains are the fix points. Note that, the lifted order re-order the uco, by ordering the elements in terms of their transformations (leaving unchanged the order among elements with the same transformation). Hence, for instance, $D_4 \sqsubseteq_{\tau_a} D_7$ (even if they are not comparable with respect to $\sqsubseteq$), since $\tau_a(D_4) = D_{11} \sqsubseteq D_7$, while $D_6 \sqsubseteq_{\tau_a} D_4$, $D_8 \sqsubseteq_{\tau_a} D_4$ and $D_{11} \sqsubseteq_{\tau_a} D_4$, precisely as it happens with $\sqsubseteq$, since all these domains have the same transformation τ_a.

Figure 1(b) provides another example of lifted order $\sqsubseteq_{\tau_b}$, where $\tau_b = \lambda X.\, X \sqcap D_3$. It is worth noting that both the domain transformers are join-uniform, implying additivity on the lifted *lco*(*Sign*), namely admitting right adjoints.

2.4. Soundness and completeness

Let $f : C \xrightarrow{\text{m}} D$ be a semantic function defined over some concrete domains C and D. Let an abstract interpretation be specified by Galois connections with abstract domains $\rho(C)$ and $\eta(D)$ corresponding to closure operators $\rho \in uco(C)$ and $\eta \in uco(D)$ respectively, and by a corresponding abstract semantics $f^\sharp : \rho(C) \xrightarrow{\text{m}} \eta(D)$. Then, $f^\sharp$ is *sound* for (or is a *correct approximation* of) f if $\eta \circ f \sqsubseteq f^\sharp \circ \rho$. This holds if and only if $\eta \circ f \circ \rho \sqsubseteq f^\sharp$. The function $\eta \circ f \circ \rho$ is called *best correct approximation of f in ρ and η*. Whenever $f : C \xrightarrow{\mathrm{m}} C$ and $f^\sharp : \rho(C) \xrightarrow{\mathrm{m}} \rho(C)$, $f^\sharp$ is *fix-point sound* for f if $\rho(\mathit{lfp}(f)) \leq \mathit{lfp}(f^\sharp)$. A sound over-approximation intuitively means that no error can be missed by the analysis, i.e., the approximate semantics includes a full coverage of all possible concrete computations, e.g., the collections of all reachable states. As we recalled in the introduction, a well-known basic result of abstract interpretation [CC79b, Theorem 7.1.0.4] states that soundness implies fix-point soundness. It is worth remarking that fix-point soundness is in general a strictly weaker property than soundness.

Precision of an abstract interpretation is typically defined in terms of *completeness* [CC79b]. Depending on where we compare the concrete and the abstract computations we obtain two different notions of completeness [GRS00, GQ01]. If we compare the results in the abstract domain, we obtain what is called *backward completeness* ($\mathscr{B}$-completeness), while, if we compare the results in the concrete domain we obtain the so called *forward completeness* ($\mathscr{F}$-completeness). Formally, if $f : C \xrightarrow{\mathrm{m}} C$ and $\rho \in \mathit{uco}(C)$, then ρ is $\mathscr{B}$-complete for f if $\rho \circ f \circ \rho = \rho \circ f$, while it is $\mathscr{F}$-complete for f if $\rho \circ f \circ \rho = f \circ \rho$. A complete over-approximation means that no false alarms are returned by the analysis, i.e., in $\mathscr{B}$-completeness the approximate semantics computed by manipulating abstract objects corresponds precisely to the abstraction of the concrete semantics, while in $\mathscr{F}$-completeness the concrete semantics does not lose precision by computing on abstract objects. The problem of making abstract domains $\mathscr{B}$-complete has been solved in [GRS00] and later extended to $\mathscr{F}$-completeness in [GQ01]. Let $f : C_1 \longrightarrow C_2$ and $\rho \in \mathit{uco}(C_2)$ and $\eta \in \mathit{uco}(C_1)$. $\langle \rho, \eta \rangle$ is a pair of $\mathscr{B}[\mathscr{F}]$-complete abstractions for f if $\rho \circ f = \rho \circ f \circ \eta$ $[f \circ \eta = \rho \circ f \circ \eta]$. A pair of domain transformers has been associated with any completeness problem, which are respectively a *domain refinement* and *simplification* [GR97]. In [GRS00] and [GQ01], a constructive characterisation of the most abstract refinement, called *complete shell*, and of the most concrete simplification, called *complete core*, of any abstract domain, making it $\mathscr{F}$ or $\mathscr{B}$-complete for a given continuous function f, is given as a solution of simple abstract domain equations given by the following basic operators:

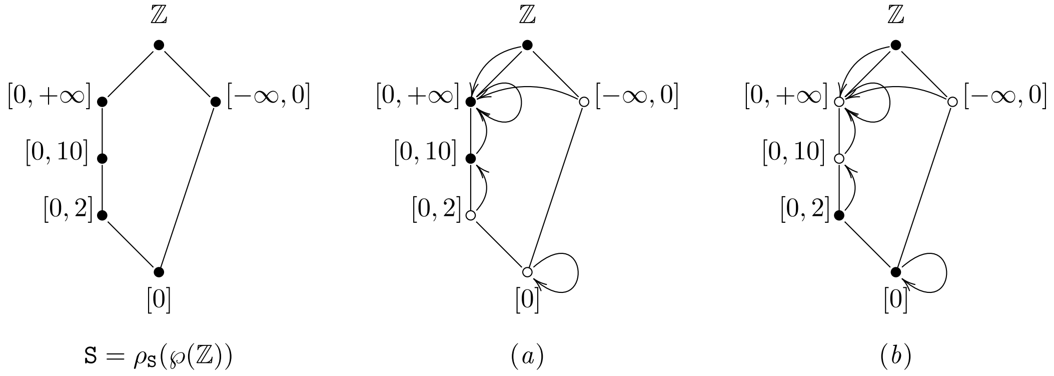
$$\boxed{\begin{array}{r|l} R_f^{\mathscr{F}} \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(f(X)) & R_f^{\mathscr{B}} \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(\bigcup_{y \in X} \max(f^{-1}(\downarrow y))) \\ C_f^{\mathscr{F}} \stackrel{\text{def}}{=} \lambda X.\, \left\{\, y \in L \,\middle|\,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

Following [GRS00], given a pair of abstract domains $\langle \rho, \eta \rangle$ and a concrete function f, the $\mathscr{B}$-complete shell of $\langle \rho, \eta \rangle$ with respect to f is the most concrete $\beta \sqsupseteq \rho$ such that $\langle \beta, \eta \rangle$ is $\mathscr{B}$-complete for f, and the $\mathscr{B}$-complete core that is the most abstract $\beta \sqsubseteq \eta$ such that $\langle \rho, \beta \rangle$ is $\mathscr{B}$-complete. It is possible to obtain the dual notions of $\mathscr{F}$-complete shell and $\mathscr{F}$-complete core by refining the output abstraction ρ and by simplifying the input abstraction η in order to gain $\mathscr{F}$-completeness. It has been proved in [GRS00]that $\mathscr{B}[\mathscr{F}]$-complete core and shell can be obtained as follows:

$$\boxed{\begin{array}{r|l} \mathscr{B}\text{-complete core: } \mathcal{C}_f^{\mathscr{B},\eta}(\rho) \stackrel{\text{def}}{=} \rho \sqcup C_f^{\mathscr{B}}(\eta) & \mathscr{B}\text{-complete shell: } \mathcal{R}_f^{\mathscr{B},\rho}(\eta) \stackrel{\text{def}}{=} \eta \sqcap R_f^{\mathscr{B}}(\rho) \\ \mathscr{F}\text{-complete core: } \mathcal{C}_f^{\mathscr{F},\rho}(\eta) \stackrel{\text{def}}{=} \eta \sqcup C_f^{\mathscr{F}}(\rho) & \mathscr{F}\text{-complete shell: } \mathcal{R}_f^{\mathscr{F},\eta}(\rho) \stackrel{\text{def}}{=} \rho \sqcap R_f^{\mathscr{F}}(\eta) \end{array}}$$

When $\eta = \rho$, we need a fix-point iteration on abstract domains. For instance $\mathcal{R}_f^{\mathscr{F}}(\rho) = \mathit{gfp}(\lambda X.\, \rho \sqcap R_f^{\mathscr{F}}(X))$ with $\mathcal{R}_f^{\mathscr{F}}(\rho) \in \mathit{lco}(\mathit{uco}(C))$ which is called *absolute $\mathscr{F}$-complete shell*. By construction if f is additive then $\mathcal{R}_f^{\mathscr{B}} = \mathcal{R}_{f^+}^{\mathscr{F}}$ [GQ01]. This means that when we have to solve a problem of $\mathscr{B}$-completeness for an additive function then we can equivalently solve the corresponding $\mathscr{F}$-completeness problem for its right adjoint. The following example from [GQ01], exemplifies the duality of forward and backward completeness.

Example 2.5. Assume $\mathtt{S}$ be the domain in Figure 2, which is an obvious abstraction of $\langle \wp(\mathbb{Z}), \subseteq \rangle$ for the analysis of integer variables and $\mathit{sq} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ be the square operation defined as follows: $\mathit{sq}(X) = \left\{\, x^2 \,\middle|\, x \in X \,\right\}$ for $X \in \wp(\mathbb{Z})$. The arrows in Figure 2 (a) and (b) show the function $\mathit{sq}^\sharp$. Let $\rho_{\mathtt{S}} \in \mathit{uco}(\wp(\mathbb{Z}))$ be the closure operator associated with $\mathtt{S}$. The best correct approximation of sq in $\mathtt{S}$ is $\mathit{sq}^\sharp : \mathtt{S} \to \mathtt{S}$ such that $\mathit{sq}^\sharp(X) = \rho_{\mathtt{S}}(\mathit{sq}(X))$, with $X \in \mathtt{S}$. It is easy to see that the abstractions $\rho_a = \{\mathbb{Z}, [0, +\infty], [0, 10]\}$ (black dots in Figure 2 (a)) and $\rho_b = \{\mathbb{Z}, [0, 2], [0]\}$ of $\mathtt{S}$ (black dots in Figure 2 (b)), respect the following facts: $\rho_a = \{\mathbb{Z}, [0, +\infty], [0, 10]\}$ is $\mathscr{F}$-complete but not $\mathscr{B}$-complete on the concrete domain $\mathtt{S}$ for $\mathit{sq}^\sharp$ (for instance $\rho_a(\mathit{sq}^\sharp(\rho_a([0]))) = [0, +\infty]$ but $\rho_a(\mathit{sq}^\sharp([0])) = [0, 10]$) and $\rho_b = \{\mathbb{Z}, [0, 2], [0]\}$ is $\mathscr{B}$-complete but not $\mathscr{F}$-complete on the concrete domain $\mathtt{S}$ for $\mathit{sq}^\sharp$ (for instance $\rho_b(\mathit{sq}^\sharp(\rho_b([0, 2]))) = \mathbb{Z}$ but $\mathit{sq}^\sharp(\rho_b([0, 2])) = [0, 10]$).



$\mathtt{S} = \rho_{\mathtt{S}}(\wp(\mathbb{Z}))$ (a) (b)

Fig. 2. The abstract domain $\mathtt{S}$ and two abstractions

$$\langle\sigma,\mathbf{skip}\rangle \Downarrow \langle\sigma,\mathbf{skip}\rangle \ \ \text{(Fix-point)} \qquad \frac{[\![\mathtt{e}]\!]\sigma = n \in \mathit{Val}}{\langle\sigma, x := \mathtt{e}\rangle \Downarrow \langle\sigma[x \mapsto n],\mathbf{skip}\rangle} \qquad \frac{\langle\sigma, st\rangle \Downarrow \sigma'}{\langle\sigma, st; \mathtt{C}_1\rangle \Downarrow \langle\sigma', \mathtt{C}_1\rangle}$$

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

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

Table 1. Small-step operational semantics of $\mathcal{L}$

2.5. Programming language and semantics

For abstract interpretation, one needs a fine-grain small-step semantics containing program points or similar syntactic information to which abstract values can be bound. Consider a simple imperative language $\mathcal{L}$:

$$\mathtt{C} \quad ::= \quad \mathbf{skip} \mid x := \mathtt{e} \mid \mathtt{C}_0; \mathtt{C}_1 \mid \mathbf{while}\ \mathtt{e}\ \mathbf{do}\ \mathtt{C}\ \ \mathbf{endw} \mid \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{C}_0\ \mathbf{else}\ \mathtt{C}_1\ \mathbf{fi}$$

A notational convenience: write **case** e **of** $v_1 : \mathtt{C}_1; \ldots ; v_n : \mathtt{C}_n$ to stand for a chain of **if** − **then** − **else** on mutually exclusive values (the v_i are the possible values that e can take, and $\mathtt{C}_i$ is the corresponding program fragment to execute). In Table 1 we consider the standard operational semantics of the language. Let $\mathbb{P}_{\mathcal{L}}$ be a set of programs in the language $\mathcal{L}$, $\mathit{Var}(\mathtt{P})$ the set of all the variables in $\mathtt{P}$ (analogously $\mathit{Var}(\mathtt{e})$ is the set of variables used in an expression $\mathtt{e}$), and $\mathbb{PL}_{\mathtt{P}}$ be a set of program points of $\mathtt{P} \in \mathbb{P}_{\mathcal{L}}$ containing a special notation ϵ for the empty program point, Val be the set of values, and $\mathbb{M} \overset{\text{def}}{=} \mathit{Var}(\mathtt{P}) \longrightarrow \mathit{Val}$ be a set of possible program memories. When a statement st belongs to a program $\mathtt{P}$ we write $st \in \mathtt{P}$, then we define the auxiliary functions $\mathtt{Stm}_{\mathtt{P}} : \mathbb{PL}_{\mathtt{P}} \to \mathbb{P}_{\mathcal{L}}$ be such that $\mathtt{Stm}_{\mathtt{P}}(l) = \mathtt{c}$ if $\mathtt{c}$ is the statement in $\mathtt{P}$ at program point l (denoted ${}^{l}\!\cdot\mathtt{c}$) and $\mathtt{Pc}_{\mathtt{P}} = \mathtt{Stm}_{\mathtt{P}}{}^{-1} : \mathbb{P}_{\mathcal{L}} \to \mathbb{PL}_{\mathtt{P}}$ with the simple extension to blocks of instructions $\mathtt{Pc}_{\mathtt{P}}(st; \mathtt{C}\,) = \mathtt{Pc}_{\mathtt{P}}(st)$ where $st \in \mathtt{P}$. Then, let $\sigma \in \mathbb{M}$, we define the semantics of $\mathcal{L}$ in Table 1, where x are variables, e are (arithmetic and boolean) expressions, $[\![\cdot]\!]$ is the evaluation of expressions, and where we write $\langle\sigma, \mathtt{C}\,\rangle \Downarrow \langle\sigma', \mathtt{C}\,'\rangle$ for the execution of $\mathtt{C}$ in the memory σ. We can formally characterise the small-step operational semantics of programs. Let $\mathbb{D} = \mathbb{M} \times \mathbb{P}_{\mathcal{L}}$ be the set of states, containing the actual memory and the code to execute, and $\langle\sigma, \mathtt{C}\,\rangle \in \mathbb{D}$. $f_{\mathcal{L}} : \mathbb{D} \longrightarrow \wp(\mathbb{D})$ is such that:

$$f_{\mathcal{L}}(\langle\sigma, \mathtt{C}\,\rangle) = \{\ \langle\sigma', \mathtt{C}\,'\rangle \mid\ \langle\sigma, \mathtt{C}\,\rangle \Downarrow \langle\sigma', \mathtt{C}\,'\rangle\ \ \}$$

It is worth noting that for deterministic programs, like $\mathcal{L}$, this set contains only one state. We abuse notation by denoting as $f_{\mathcal{L}}$ also its trivial additive lift on $\wp(\mathbb{D})$. We define the small-step program semantics as the fix-point of the transfer function $f_{\mathcal{L}}$ starting from a set of initial states $S \in \wp(\mathbb{D})$: $[\![\mathtt{P}]\!](S) \overset{\text{def}}{=} \mathit{lfp}_S f_{\mathcal{L}} \in \wp(\mathbb{D})$.

In the following, when we consider the semantics of a program P starting from any possible initial memory state of P we simply write $[\![\mathtt{P}]\!]$, denoting the set $\{\ \mathit{lfp}_{\langle\sigma,\mathtt{P}\rangle} f_{\mathfrak{L}} \mid \ \sigma \in \mathbb{M}\ \}$.

3. Making abstract interpretations incomplete

As proved in [GRS00], completeness is a property concerning uniquely the abstract domain and the (concrete semantics of the) program to be analysed (see [GLR15] for a recent account on proving abstract interpretations completeness). Therefore,f make an abstract interpretation complete (respectively incomplete) we may only act on the abstract domain (e.g., by abstraction refinements) or by code refactoring (see [LL09] for an example of code transformations that improves the precision of given analyses). Our aim is to model the potency of an (obfuscated) program P. Therefore the program here is fixed, and understanding the potency of the obfuscations employed in P means understanding what makes an abstract domain imprecise (viz., incomplete) for P. This corresponds precisely to remove all the elements in the abstract domain that may be introduced by the completeness refinement for P. Following this observation, we introduce the idea of *incomplete domain compressor*: The most abstract domain having a given complete refinement, namely the right adjoint of the complete shell refinement viewed as an abstract domain transformer. In this section we prove that the incomplete compressor exists under weak hypotheses and that it induces incomplete abstract interpretations for programs in our simple imperative programming language.

3.1. Simplifying abstractions

In the following, we show that a complete shell always admits a right adjoint. Indeed, by Proposition 2.3 the right adjoint of an lco exists if and only if the lco is join-uniform. At this point, since complete shells have the form of pattern completion we show that pattern completion domain transformers are always join-uniform.

Lemma 3.1. Let C be a complete lattice and $\eta \in \mathit{uco}(C)$ then the pattern completion function $f_\eta \overset{\text{def}}{=} \lambda\delta.\delta \sqcap \eta$ is join-uniform.

Proof. We have to prove that $f_\eta(\sqcup\{\ \delta \in \mathit{uco}(C) \mid\ f_\eta(\delta) = f_\eta(\rho)\ \}) = f_\eta(\rho)$. In other words we have to prove that $\sqcup\{\ \delta \in \mathit{uco}(C) \mid\ f_\eta(\delta) = f_\eta(\rho)\ \} \sqcap \eta = \rho \sqcap \eta$.
In the sake of simplicity, let $\{\delta_i\}_{i\in Z} \overset{\text{def}}{=} \{\ \delta \in \mathit{uco}(C) \mid\ f_\eta(\delta) = f_\eta(\rho)\ \}$, then we want to prove that if $\forall i,j \in Z.\delta_i \sqcap \eta = \delta_j \sqcap \eta$ then $(\bigsqcup_{i\in Z} \delta_i) \sqcap \eta = \delta_j \sqcap \eta$. Note that, $\delta_j \sqsubseteq \bigsqcup_{i\in Z} \delta_i$ hence the $\sqsubseteq$ relation holds. We have to prove the other inclusion, namely that $\forall x \in \delta_j \sqcap \eta$ then $x \in (\bigsqcup_{i\in Z} \delta_i) \sqcap \eta$.
First of all let us note that if $M_i \overset{\text{def}}{=} \mathit{Mirr}(\delta_i \sqcap \eta)$ then $\forall j.M_i \subseteq \delta_j \sqcap \eta$. This implies that $\forall j.\bigcup_{i\in Z} M_i \subseteq \delta_j \sqcap \eta$. But then $\forall y \in \bigcup_{i\in Z} M_i$ we have $y \in \delta_j$ or $y \in \eta$, being y meet-irreducible, which implies that $\forall y \in (\bigcup_{i\in Z} M_i) \smallsetminus \eta$ we have $y \in \bigcap_{i\in Z} \delta_i$. At this point the following implication holds

$$\begin{array}{rcl}
\forall i \in Z.\, x & \in & \delta_i \sqcap \eta \Rightarrow \exists Y \subseteq \mathit{Mirr}(\delta_i \sqcap \eta) \text{ s.t. } \bigwedge Y = x \\
& \Rightarrow & \exists Y \subseteq \bigcup_{i\in Z} M_i \text{ s.t. } \bigwedge Y = x \\
& \Rightarrow & \exists Y \subseteq \bigcup_{i\in Z} M_i,\ \ Y \smallsetminus \eta \subseteq \bigcup_{i\in Z} M_i \smallsetminus \eta \subseteq \bigcap_{i\in Z} \delta_i \text{ s.t. } \bigwedge Y = x \\
& \Rightarrow & \exists Y \subseteq \eta \cup \bigcap_{i\in Z} \delta_i \text{ s.t. } \bigwedge Y = x \\
& \Rightarrow & x = \bigwedge Y \in \mathcal{M}(\eta \cup \bigcap_{i\in Z} \delta_i) = \mathcal{M}(\eta \cup \bigsqcup_{i\in Z} \delta_i) = (\bigsqcup_{i\in Z} \delta_i) \sqcap \eta
\end{array}$$

□

Note that, the domain transformers defined in Example 2.4 are exactly of this form, and indeed, the fact that they admit right adjoint on the lifted orders depends precisely on the fact that these transformers are join-uniform by Lemma 3.1.

3.1.1. Forward incomplete compressor

Consider $\mathscr{F}$-completeness, i.e., $\rho \circ f \circ \eta = f \circ \eta$ with $\rho, \eta \in \mathit{uco}(C)$, C complete lattice, and $f : C \longrightarrow C$, denoting also its additive lift to $\wp(C)$. The complete shell is $\mathcal{R}^{\mathscr{F}}_{f,\eta}$ which refines the output domain by adding all the f-images of elements of η to ρ. Hence, by Lemma 3.1, we have the following result.

Proposition 3.2. $\mathcal{R}^{\mathcal{F}}_{f,\eta} = \lambda\rho.\, \rho \sqcap \mathcal{M}(f(\eta))^2$ is join-uniform on $uco(C)$.

Proof. Trivially by Lemma 3.1 $\square$

Being $\mathcal{R}^{\mathcal{F}}_{f,\eta}$ join-uniform, its right adjoint exists (Proposition 2.3). We show that, under specific hypotheses, the right adjoint can be characterised as the following transformer:

$$\mathcal{UR}^{\mathcal{F}}_{f,\eta} \overset{\text{def}}{=} \lambda\rho.\, \mathcal{M}(\mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) \smallsetminus \mathcal{M}(f(\eta)))$$

This transformation first erases all the elements that we should have to avoid if we want to loose precision for the computation of function f, and then by the Moore-family completion adds only those necessary for obtaining a Moore-family, i.e., an abstract domain. We call this transformation *incomplete compressor.* We first prove a lemma providing a necessary results for the following proposition. This lemma is a particular case of the result in [FR96] which allows us to remove the hypothesis of meet-generation necessary in general for characterising pseudo-complement as set difference.

Lemma 3.3. Let $\alpha, \beta \in uco(C)$ be abstract domains, then $\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$ is the most abstract domain such that $\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta = \alpha \sqcap \beta$, i.e.,

$$\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) = (\alpha \sqcap \beta) \ominus \beta.$$

Proof. Let us first prove that $\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta = \alpha \sqcap \beta$. It is clear that one inclusion is trivial, since if $x \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$ then it must be in $\mathit{Mirr}(\alpha \sqcap \beta) \subseteq \alpha \sqcap \beta$.
Let us prove the other inclusion. Suppose $x \in \alpha \sqcap \beta$.

- Let $x \in \mathit{Mirr}(\alpha \sqcap \beta)$ and $x \in \alpha \smallsetminus \beta$[3]. Then $x \in \mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta$, and therefore $x \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta$;
- Let $x \in \mathit{Mirr}(\alpha \sqcap \beta)$ and $x \in \beta$. Then trivially it is in any set in product with β, i.e., $\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta$;
- Let $x \notin \mathit{Mirr}(\alpha \sqcap \beta)$. Then there exists $Z \subseteq \mathit{Mirr}(\alpha \sqcap \beta)$ such that $\bigwedge Z = x$. Let $(Z \cap \alpha) \smallsetminus \beta = Z_1$ and $Z \cap \beta = Z_2$.
  - If $Z_2 = \varnothing$ then $Z \subseteq \alpha$ and $Z \cap \beta = \varnothing$ implying that $Z \subseteq \mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta$, hence $x = \bigwedge Z \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta$;
  - If $Z_2 \neq \varnothing$, being Z a set of meet-irreducible any of its element can be generated in $\alpha \sqcap \beta$ by meet, hence all its elements are either in α or in β, i.e., $Z = Z_1 \cup Z_2$.Then $Z_1 \subseteq \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, being a set of meet-irreducible elements of α, while $Z_2 \subseteq \beta$, hence $Z = Z_1 \cup Z_2 \subseteq \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \cup \beta$. Namely, $x = \bigwedge Z \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta$.

Let us prove now that it is the most abstract domain with this property. Namely, suppose there exists ρ such that $\rho \sqcap \beta = \alpha \sqcap \beta$, we prove that $\rho \sqsubseteq \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$. Let us consider $x \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, we prove that $x \in \rho$. The hypothesis on x implies that $x \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqcap \beta = \alpha \sqcap \beta = \rho \sqcap \beta$, namely $x \in \rho \sqcap \beta$ and $x \in \alpha \sqcap \beta$.

- Suppose $x \in \mathit{Mirr}(\alpha \sqcap \beta)$. Then, since $x \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, being x meet-irreducible, we have $x \in \mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta$, namely $x \notin \beta$. At this point, being $\alpha \sqcap \beta = \rho \sqcap \beta$ we have also that $x \in \mathit{Mirr}(\rho \sqcap \beta)$, namely $x \in \rho$ or $x \in \beta$, but we have just proved that $x \notin \beta$, hence $x \in \rho$.
- Let $x \notin \mathit{Mirr}(\alpha \sqcap \beta)$, then $x \notin \mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta$, but by hypothesis $x \in \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, hence there exists $Z \subseteq \alpha$, with $Z \cap \beta = \varnothing$, such that $\bigwedge Z = x$. Finally, $Z \subseteq \alpha \sqcap \beta = \rho \sqcap \beta$ with Z set of meet-irreducible that are not in β. Hence, Z must be subset of ρ, which implies that also $x = \bigwedge Z \in \rho$.

$\square$

Proposition 3.4. $\mathcal{UR}^{\mathcal{F}}_{f,\eta} = (\mathcal{R}^{\mathcal{F}}_{f,\eta})^+$.

Proof. By Proposition 3.2, we have that $\mathcal{R} \overset{\text{def}}{=} \mathcal{R}^{\mathcal{F}}_{f,\eta}$ is join-uniform. Hence, by Proposition 2.2, we can characterise its right adjoint as

$$\mathcal{R}^+ = \lambda\rho.\, \bigsqcup\{\ \delta \mid\ \mathcal{R}(\delta) = \mathcal{R}(\rho)\ \ \} = \lambda\rho.\, \bigsqcup\{\ \delta \mid\ \ \delta \sqcap \mathcal{M}(f(\eta)) = \rho \sqcap \mathcal{M}(f(\eta))\ \ \}$$

[2] $f(\eta)$ stands for $f(\eta(C))$

[3] Note that if $x \in \mathit{Mirr}(\alpha \sqcap \beta)$, then x is meet-irreducible, therefore it must be in α or in β.

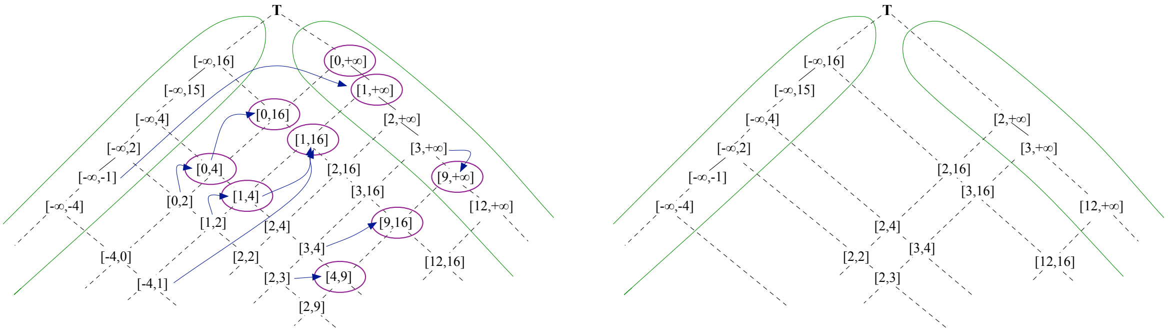


Fig. 3. Abstract domain and transformation of Example 3.5

By join-uniformity we know that $\mathcal{R} \circ \mathcal{R}^+(\rho) = \mathcal{R}(\rho)$. Hence

$$\mathcal{R}^+(\rho) \in \{ \ \delta \mid \ \delta \sqcap \mathcal{M}(f(\eta)) = \rho \sqcap \mathcal{M}(f(\eta)) \ \ \}$$

and by definition of $\mathcal{R}^+$ this means that it is the most abstract domain such that $\mathcal{R}^+(\rho) \sqcap \mathcal{M}(f(\eta)) = \rho \sqcap \mathcal{M}(f(\eta))$. At this point we can observe that this is precisely the pseudo-complement $(\rho \sqcap \mathcal{M}(f(\eta))) \ominus \mathcal{M}(f(\eta))$[4]. By Lemma 3.3 we conclude that $(\rho \sqcap \mathcal{M}(f(\eta))) \ominus \mathcal{M}(f(\eta)) = \mathcal{M}(\mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) \smallsetminus \mathcal{M}(f(\eta)))$. □

Example 3.5. Consider the example in Figure 3 on the left. The square operation $sq(X) = \{ \ x^2 \mid \ x \in X \ \ \}$ for $X \in \wp(\mathbb{Z})$ is depicted with arrows on the lattice of integer intervals Int, which is defined as usual as $\mathit{Int} \overset{\text{def}}{=} \{ \ [a,b] \mid \ a,b \in \mathbb{Z} \ \ \} \cup \{ \ [-\infty,b] \mid \ b \in \mathbb{Z} \ \ \} \cup \{ \ [a,+\infty] \mid \ a \in \mathbb{Z} \ \ \}$ [CC77]. In this case, the best correct approximation of sq in Int is $sq^\sharp : \mathit{Int} \to \mathit{Int}$ such that $sq^\sharp(X) = \mathit{Int}(sq(X))$, with $X \in \mathit{Int}$. Note that, by definition of $sq^\sharp$, we trivially have $\mathit{Int} \circ sq^\sharp \circ \mathit{Int} = sq^\sharp \circ \mathit{Int}$, i.e., $\mathscr{F}$-completeness. For instance $sq^\sharp([3,4]) = [9,16] \in \mathit{Int}$. Let us transform the output Int domain in order to induce incompleteness, namely let us derive the forward incomplete compression of Int. Note that, $\mathit{Mirr}(\mathit{Int}) = \{ \ [-\infty,b] \mid \ b \in \mathbb{Z} \ \ \} \cup \{ \ [a,+\infty] \mid \ a \in \mathbb{Z} \ \ \}$ [GRS00], in the picture collected in the open lines, and that

$$\mathcal{M}(sq^\sharp(\mathit{Int})) = \{ \ [a^2,b^2] \mid \ a,b \in \mathbb{Z} \ \ \} \cup \{ \ [a^2,+\infty] \mid \ a \in \mathbb{Z} \ \ \} \sqsupseteq \mathit{Int}.$$

are depicted with circled lines.
Hence, we have that $\mathit{Int}' \overset{\text{def}}{=} \mathcal{UR}^{\mathscr{F}}_{sq^\sharp,\mathit{Int}}(\mathit{Int}) = \mathcal{M}(\mathit{Mirr}(\mathit{Int}) \smallsetminus \mathcal{M}(sq^\sharp(\mathit{Int}))$ namely

$$\begin{aligned} \mathit{Int}' &= \mathcal{M}(\{ \ [-\infty,b] \mid \ b \in \mathbb{Z} \ \ \} \cup \{ \ [a,+\infty] \mid \ a \in \mathbb{Z}, \nexists c \in \mathbb{Z}.\ a = c^2 \ \ \}) \\ &= \{ \ [a,b] \mid \ a,b \in \mathbb{Z}, \nexists c,d \in \mathbb{Z}.\ a = c^2 \ \vee \ b = d^2 \ \ \} \cup \\ &\quad\ \{ \ [-\infty,b] \mid \ b \in \mathbb{Z} \ \ \} \cup \{ \ [a,+\infty] \mid \ a \in \mathbb{Z}, \nexists c \in \mathbb{Z}.\ a = c^2 \ \ \} \end{aligned}$$

depicted on the right in Figure 3. So, for instance, we have that $sq^\sharp([3,4]) = [9,16] \notin \mathit{Int}'$, meaning incompleteness.

Note that, this transformation does not always generate an incomplete domain. The following result provides the formal conditions that have to hold in order to induce incompleteness, namely in order to guarantee the existence of incomplete compression. The domains that does not satisfy these conditions are complete and are complete shells of only themselves, namely we cannot find a unique most concrete simplification which is incomplete.

Theorem 3.6. Let $\eta, \rho \in \mathit{uco}(C)$ and $f : C \longrightarrow C$. $\mathcal{UR}^{\mathcal{F}}_{f,\eta}(\rho)$ (here denoted $\mathcal{UR}$) is such that $\mathcal{UR}(\rho) \circ f \circ \eta \neq f \circ \eta$ if and only if one of the following conditions hold:

1. $\rho \circ f \circ \eta \neq f \circ \eta$, i.e., ρ was incomplete before simplification;
2. $\mathcal{M}(f(\eta)) \cap \mathit{Mirr}(\rho) \neq \varnothing$;

[4] If C is a meet-semilattice with bottom, then the pseudo-complement of $x \in C$, when it exists, is the unique element $x^* \in C$ such that $x \wedge x^* = \bot$ and such that $\forall y \in C.\ (x \wedge y = \bot) \Rightarrow (y \leq x^*)$.

Proof. (⇒) Suppose $(*)$ $\mathit{Mirr}(\rho) \cap \mathcal{M}(f(\eta)) = \varnothing$ and suppose $(**)$ $\rho \circ f \circ \eta = f \circ \eta$. Note that, by [GRS00] we know that if $\rho \circ f \circ \eta = f \circ \eta$ then $\rho \sqsubseteq \mathcal{M}(f(\eta))$. Then we have $\rho \sqcap \mathcal{M}(f(\eta)) = \rho$, hence $\mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) = \mathit{Mirr}(\rho)$. But then, by hypothesis $(**)$ we have that $\mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) \smallsetminus \mathcal{M}(f(\eta)) = \mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\eta)) = \mathit{Mirr}(\rho)$, namely $\mathcal{UR}(\rho) = \rho$. At this point, we also trivially have that $\mathcal{UR}(\rho) \circ f \circ \eta = f \circ \eta$.

(⇐) Suppose $\mathcal{UR}(\rho) \circ f \circ \eta = f \circ \eta$, by [GRS00] this means that $\mathcal{UR}(\rho) \sqsubseteq \mathcal{M}(f(\eta))$. Hence

$$\begin{aligned} \mathcal{M}(f(\eta)) &\sqsupseteq \mathcal{UR}(\rho) = \mathcal{M}(\mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) \smallsetminus \mathcal{M}(f(\eta))) \\ &\sqsupseteq \mathcal{M}(\mathit{Mirr}(\rho)) = \rho \end{aligned}$$

since $\mathit{Mirr}(\rho) \subseteq \mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) \smallsetminus \mathcal{M}(f(\eta))$, namely (again by [GRS00]) we have that $\rho \circ f \circ \eta = f \circ \eta$. At this point, this last condition implies that $\mathcal{UR}(\rho) = \mathcal{M}(\mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\eta)))$. Hence, if $\exists\, x \in \mathcal{M}(f(\eta)) \cap \mathit{Mirr}(\rho) \neq \varnothing$, but then we would have that $x \notin \mathcal{UR}(\rho)$ (since being $x \in \mathit{Mirr}(\rho)$ cannot be generated by $\mathcal{M}$ starting from a subset of $\mathit{Mirr}(\rho)$). Moreover, $x \in f(\eta)$ because x is meet irreducible in ρ and $\mathcal{M}(f(\eta)) \subseteq \rho$, hence x cannot be generated by $\mathcal{M}$ also in $f(\eta)$. But this would imply that on $x \in f(\eta)$ we have that $\mathcal{UR}(\rho)$ is not complete, which is against the hypothesis. Therefore we also have that $\mathcal{M}(f(\eta)) \cap \mathit{Mirr}(\rho) = \varnothing$. □

In the following examples, we show the meaning of these conditions.

Example 3.7. Consider the *Sign* domain in Figure 1. Consider a complete shell such that $\mathcal{M}(f(\eta)) = D_7$, then the completeness transformer is $\mathcal{R} = \lambda X.X \sqcap D_7$. The resulting lco on the corresponding lifted order is in Figure 1(a), where the circled domains are the complete ones, i.e., $\{D_7, D_{11}, D_{12}, D_{14}\}$. All of them contain the meet-irreducible elements of D_7 (condition (2) of Theorem 3.6 is satisfied) and therefore we can find the incomplete compression of any domain, e.g., $\mathcal{UR}(D_{12}) = D_5$.

Theorem 3.6 says that some conditions have to hold in order to have a *unique* incomplete simplification, this does not mean that we cannot find anyway an incomplete simplification, even if it is not unique. Consider the following example.

Example 3.8. Consider again the domain in Figure 1 and suppose the shell now is $\mathcal{R} = \lambda X.X \sqcap D_3$. The lifted lco is depicted in Figure 1(b). In this case the complete domains are $\{D_3, D_7, D_8, D_9, D_{11}, D_{12}, D_{13}, D_{14}\}$. We can observe that not all of them have meet-irreducibles in common with D_3. In particular, D_{12} and D_{14} are shell only of themselves. In this case, we could only choose one of the closest complete domains that contains meet-irreducible elements of D_3, e.g., for D_{14} we can choose between D_{11} or D_{13}, and then we can transform one of the chosen domains for finding one of the closest incomplete domains, i.e., D_6 or D_{10}.

Absolute incomplete compressor. We can exploit the previous transformation relative to a starting input abstraction ρ, in order to characterise the abstract domain which is incomplete for a given function, both in input and in output. This is possible without fix-point iteration since, the domain transformer reaches the fix-point in one shot. The following lemma provides a property needed for proving the following theorem.

Lemma 3.9. Let $\alpha, \beta \in \mathit{uco}(C)$, then we have that $\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) = \alpha$ iff $\mathit{Mirr}(\alpha) \subseteq \mathit{Mirr}(\alpha \sqcap \beta)$ and $\beta \cap \mathit{Mirr}(\alpha) = \varnothing$.

Proof. Note that, by construction $\mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta) \sqsupseteq \alpha$, we have to prove the other inclusion. Suppose (1) $\mathit{Mirr}(\alpha) \subseteq \mathit{Mirr}(\alpha \sqcap \beta)$ and (2) $\beta \cap \mathit{Mirr}(\alpha) = \varnothing$. We observe that condition (1) implies that $\mathit{Mirr}(\alpha) \smallsetminus \beta \subseteq \mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta$, but by condition (2) we have that $\mathit{Mirr}(\alpha) = \mathit{Mirr}(\alpha) \smallsetminus \beta$. Hence $\alpha = \mathcal{M}(\mathit{Mirr}(\alpha)) \subseteq \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, namely $\alpha \sqsupseteq \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$. Therefore, we have that (1) and (2) implies the equality.
We have to prove now the other implication. If condition (2) does not hold then $\alpha = \mathcal{M}(\mathit{Mirr}(\alpha)) \subset \mathcal{M}(\mathit{Mirr}(\alpha) \smallsetminus \beta) \subseteq \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, if condition (1) does not hold then $\mathcal{M}(\mathit{Mirr}(\alpha) \smallsetminus \beta) \subset \mathcal{M}(\mathit{Mirr}(\alpha \sqcap \beta) \smallsetminus \beta)$, in any case we cannot have the equality. □

Theorem 3.10. Let $f : C \longrightarrow C$ be a monotone function, $\rho \in \mathit{uco}(C)$.
Let $\mathcal{UR}(\rho) \overset{\text{def}}{=} \mathcal{UR}^{\mathscr{F}}_{f,\rho}(\rho) \in \mathit{uco}(C)$ be an incomplete compression of ρ such that we have $\mathcal{UR}(\rho) \neq \top$. Then $\mathcal{UR}(\rho) \circ f \circ \mathcal{UR}(\rho) \neq f \circ \mathcal{UR}(\rho)$.

Proof. If $\mathcal{UR}$ is an incomplete compression then the conditions of Theorem 3.6 hold and $\mathcal{UR}(\rho) \circ f \circ \rho \neq f \circ \rho$. Let us prove that $\mathcal{UR}$ is idempotent, namely $\mathcal{UR}(\mathcal{UR}(\rho)) = \mathcal{UR}(\rho)$. Let $\rho_1 \overset{\text{def}}{=} \mathcal{UR}(\rho) \neq \top$, now we want to find the simplification of ρ_1 that makes $\mathcal{UR}(\rho_1) \circ f \circ \rho_1 \neq f \circ \rho_1$ to hold. This corresponds to use the new abstract

domain ρ_1 as the input domain, and compute the abstraction of ρ_1 in output inducing incompleteness. Let us prove that $\mathcal{UR}(\rho_1) = \rho_1$. Consider Lemma 3.9, where $\alpha = \rho_1$ and $\beta = \mathcal{M}(f(\rho_1))$. Then we have that $\mathcal{UR}(\rho_1) \overset{\text{def}}{=} \mathcal{M}(\mathit{Mirr}(\rho_1 \sqcap \mathcal{M}(f(\rho_1))) \smallsetminus \mathcal{M}(f(\rho_1))) = \rho_1$ iff (1) $\mathit{Mirr}(\rho_1) \subseteq \mathit{Mirr}(\rho_1 \sqcap \mathcal{M}(f(\rho_1)))$ and (2) $\mathcal{M}(f(\rho_1)) \cap \mathit{Mirr}(\rho_1) = \varnothing$.
Let us prove (1). Note that, $\mathit{Mirr}(\rho_1) = \mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\rho))$, hence we have to prove that $\mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\rho)) \subseteq \mathit{Mirr}(\rho_1 \sqcap \mathcal{M}(f(\rho_1)))$. Suppose, ad absurdum, that there exists $x \in \mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\rho))$ (then $x \in \rho_1$ and $x \in \mathit{Mirr}(\rho)$, i.e., $\nexists y_1, y_2 \in \rho.\ x = y_1 \wedge y_2$) such that $x \notin \mathit{Mirr}(\rho_1 \sqcap \mathcal{M}(f(\rho_1)))$. The first condition implies that $x \notin \mathcal{M}(f(\rho))$ and, by monotonicity $x \notin \mathcal{M}(f(\rho_1))$, since $\mathcal{M}(f(\rho_1)) \sqsupseteq \mathcal{M}(f(\rho))$. Now since $x \in \rho_1$ we have $x \in \rho_1 \sqcap \mathcal{M}(f(\rho_1))$ but by hypothesis we have also that $x \notin \mathit{Mirr}(\rho_1 \sqcap \mathcal{M}(f(\rho_1)))$, hence there exist $y_1, y_2 \in \rho_1 \sqcap \mathcal{M}(f(\rho_1))$ such that $x = y_1 \wedge y_2$. We cannot have $y_1, y_2 \in \rho_1$ since $x \in \mathit{Mirr}(\rho_1)$, and we cannot have $y_1, y_2 \in \mathcal{M}(f(\rho_1))$ since otherwise $x \in \mathcal{M}(f(\rho_1))$. Hence, $y_1 \in \rho_1$ and $y_2 \in \mathcal{M}(f(\rho_1))$. These imply that $y_1 \in \rho \sqsubseteq \rho_1$ and $y_2 \in \mathcal{M}(f(\rho)) \sqsubseteq \mathcal{M}(f(\rho_1))$, but this imply that x is not meet-irreducible in $\rho \sqcap \mathcal{M}(f(\rho))$, hence by construction x cannot be meet-irreducible in ρ_1. Therefore, condition (1) holds. Consider now (2), then for what we observed before $x \in \mathcal{M}(f(\rho_1))$ implies $x \in \mathcal{M}(f(\rho))$, which implies $x \notin \mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\rho))$. On the other hand, if $x \in \mathit{Mirr}(\rho) \smallsetminus \mathcal{M}(f(\rho))$ then $x \notin \mathcal{M}(f(\rho))$ which implies $x \notin \mathcal{M}(f(\rho_1))$. □

Note that, if $\mathcal{UR}(\rho) = \top$ we cannot find the absolute incomplete compressor since $\top \circ f \circ \top = \top \circ f$ always holds.

Example 3.11. Consider the situation described in Example 3.5, and compute the absolute incomplete compressor $\mathcal{UR}^{\mathscr{F}}_{sq^\sharp,\mathit{Int}'}(\mathit{Int}')$. We show that, as stated in Theorem 3.10 , the fix point is reached at the first step. Recall that:

$$\mathit{Mirr}(\mathit{Int}') = \left\{\ [-\infty, b] \,\middle|\ b \in \mathbb{Z}\ \right\} \cup \left\{\ [a, +\infty] \,\middle|\ a \in \mathbb{Z}, \nexists c \in \mathbb{Z}.\ a = c^2\ \right\}$$

$$\begin{gathered}\mathcal{M}(sq^\sharp(\mathit{Int}')) = \left\{\ [a^2, b^2] \,\middle|\ a, b \in \mathbb{Z}, \nexists c, d \in \mathbb{Z}.\ a = c^2\ \vee\ b = d^2\ \right\} \cup \\ \left\{\ [-\infty, b^2] \,\middle|\ b \in \mathbb{Z}\ \right\} \cup \left\{\ [a^2, +\infty] \,\middle|\ a \in \mathbb{Z}, \nexists c \in \mathbb{Z}.\ a = c^2\ \right\}\end{gathered}$$

Now we show that Theorem 3.10 holds. Observe that $\mathit{Mirr}(\mathit{Int}') \subseteq \mathit{Mirr}(\mathit{Int}' \sqcap \mathcal{M}(sq^\sharp(\mathit{Int}'))$, since by construction if $x \in \mathit{Mirr}(\mathit{Int}')$ then we also have $x \in \mathit{Mirr}(\mathit{Int})$, on the other hand $\mathcal{M}(sq^\sharp(\mathit{Int}')) \subseteq \mathit{Int}$, therefore x remain meet-irreducible also in the reduced product. Therefore,

$$\mathit{Mirr}(\mathit{Int}') = \mathit{Mirr}(\mathit{Int}') \smallsetminus \mathcal{M}(sq^\sharp(\mathit{Int}')) \subseteq \mathit{Mirr}(\mathit{Int}' \sqcap \mathcal{M}(sq^\sharp(\mathit{Int}')) \smallsetminus \mathcal{M}(sq^\sharp(\mathit{Int}'))$$

namely $\mathit{Int}' \sqsupseteq \mathcal{UR}^{\mathscr{F}}_{sq^\sharp,\mathit{Int}'}(\mathit{Int}')$, and since by construction we have the other inclusion, we showed the equality, i.e., $\mathcal{UR}^{\mathscr{F}}_{sq^\sharp,\mathit{Int}'}(\mathit{Int}') = \mathit{Int}'$.

Example 3.12. Consider the ρ_b domain in Figure 2(b).
Then $\mathit{Mirr}(\rho_b) = \{[0, +\infty], [-9, 0], [0, 9]\}$ and $\mathcal{M}(sq^{\mathtt{S}}(\rho_b)) = \{\mathbb{Z}, [0, +\infty], [0, 99], [0]\}$.

$$\begin{aligned}\mathtt{S}' &\overset{\text{def}}{=} \mathcal{UR}^{\mathscr{F}}_{sq^{\mathtt{S}},\rho_b}(\rho_b) = \mathcal{M}(\mathit{Mirr}(\rho_b \sqcap \mathcal{M}(sq^{\mathtt{S}}(\rho_b))) \smallsetminus \mathcal{M}(sq^{\mathtt{S}}(\rho_b))) \\ &= \mathcal{M}(\mathit{Mirr}(\rho_b) \smallsetminus \mathcal{M}(sq^{\mathtt{S}}(\rho_b))) = \mathcal{M}(\{[0, +\infty], [-9, 0], [0, 9]\}) = \{\mathbb{Z}, [-9, 0], [0, 9], [0]\}\end{aligned}$$

Finally, we can easily check that $\mathtt{S}' \circ sq^{\mathtt{S}} \circ \mathtt{S}' \neq sq^{\mathtt{S}} \circ \mathtt{S}'$.

3.1.2. Backward incompleteness compressor

In this section, we show that all the results holding for $\mathscr{F}$-completeness can be instantiated also to $\mathscr{B}$-completeness. First of all, by Lemma 3.1 we have that

Proposition 3.13. $\mathcal{R}^{\mathscr{B}}_{f,\rho}$ is join-uniform on $uco(C)$.

Proof. Trivially by Lemma 3.1. □

This result tells us that also the $\mathscr{B}$ shell admits right adjoint, and as before, its adjoint can be characterised as a pseudo-complement in the following way.

Proposition 3.14. Let $R_f \overset{\text{def}}{=} \lambda\delta.\ \mathcal{M}(\bigcup_{y \in \delta} \max(f^{-1}(\downarrow y))) \in uco(C)$, then we have that

$$\mathcal{UR}^{\mathscr{B}}_{f,\rho} \overset{\text{def}}{=} \lambda\eta.\ \mathcal{M}(\mathit{Mirr}(\eta \sqcap R_f(\rho)) \smallsetminus R_f(\rho)) = (\mathcal{R}^{\mathscr{B}}_{f,\rho})^+.$$

Proof. Analogous to Proposition 3.4. □

Finally, also for $\mathscr{B}$-completeness we can prove that the $\mathscr{B}$-incomplete compressor exists iff some conditions hold, as stated in the following theorem.

Theorem 3.15. Let $\eta, \rho \in \mathit{uco}(C)$ and $f : C \longrightarrow C$. $\mathcal{UR}^{\mathscr{B}}_{f,\rho}(\eta)$ (here denoted simply $\mathcal{UR}$) is such that $\rho \circ f \circ \mathcal{UR}(\eta) \neq \rho \circ f$ iff one of the following conditions hold:

1. $\rho \circ f \circ \eta \neq \rho \circ f$, i.e., η was incomplete before simplification;
2. $R_f(\rho) \cap \mathit{Mirr}(\eta) \neq \varnothing$;

Proof. Analogous to Theorem 3.6. □

Finally, we can characterise also the absolute $\mathscr{B}$-incomplete compressor.

Theorem 3.16. Let $f : C \longrightarrow C$ be a monotone function, $\eta \in \mathit{uco}(C)$.
Let $\mathcal{UR}(\eta) \overset{\text{def}}{=} (\mathcal{R}^{\mathscr{B}}_{f,\eta})^{+}(\eta) \in \mathit{uco}(C)$ be an incomplete compressor such that we have $\mathcal{UR}(\eta) \neq \top$. Then $\mathcal{UR}(\eta) \circ f \circ \mathcal{UR}(\eta) \neq \mathcal{UR}(\eta) \circ f$.

Proof. Analogous to Theorem 3.10. □

3.2. Refining abstractions: Incomplete expanders

If we consider the other direction, when we want to transform the input abstraction, it is well known that, for inducing $\mathscr{F}$ [$\mathscr{B}$] completeness we can simplify the domain by erasing all the η-elements whose f [inverse] image goes out of ρ. In this case, we are considering the completeness core $\mathcal{C}^{\mathscr{F}}_{\rho,f}$ [$\mathcal{C}^{\mathscr{B}}_{\eta,f}$]. If we aim at inducing incompleteness we should add all the elements such that the f [inverse] image is out of ρ, i.e., $\{\ x \mid\ f(x) \notin \rho\ \}$ [$\{\ y \mid\ \max\{\ x \mid\ f(x) \leq y\ \} \not\subseteq \eta\ \}$]. We wonder whether this transformation always exists.
Unfortunately, the following result implies, by Proposition 2.3, that we cannot find the most concrete abstraction that refines ρ and which is incomplete.

Theorem 3.17. The operator $\mathcal{C}^{\mathscr{F}}_{\rho,f}$ [$\mathcal{C}^{\mathscr{B}}_{\eta,f}$] is not meet-uniform.

Proof. Let us consider the closure $\eta \overset{\text{def}}{=} \{\top, x, y, x \wedge y, \bot\}$ and $\rho \in \mathit{uco}$, suppose $\{f(\top), f(x \wedge y)\} \subseteq \rho$ and suppose $\{f(x), f(y), f(\bot)\} \cap \rho = \varnothing$. Then $\mathcal{C}(\eta) = \{\ z \mid\ f(z) \in \rho\ \} = \{\top, x \wedge y\}$.
Consider now the following abstractions of η: $\delta_1 \overset{\text{def}}{=} \{\top, x\}$ and $\delta_2 \overset{\text{def}}{=} \{\top, y\}$, we have that $\mathcal{C}(\delta_1) = \mathcal{C}(\delta_2) = \{\top\} \neq \mathcal{C}(\delta_1 \sqcap \delta_2) = \mathcal{C}(\{\top, x, y, x \wedge y\}) = \{\top, x \wedge y\}$. □

4. Modelling the potency of code obfuscation

In this section, we show how the theoretical results described in the previous section can be used in the field of code obfuscation in order to certify the potency of an obfuscator and to provide insights on how to build an obfuscator that defeats a given attacker. To this end, we consider a program $\mathtt{P} \in \mathbb{P}_{\mathcal{L}}$ and its denotational, i.e., I/O, semantics $[\![\mathtt{P}]\!]$, computed as fix-point of the language interpreter operation $f_{\mathcal{L}}$, namely $[\![\mathtt{P}]\!] = \{\ \mathit{lfp}_{\langle\sigma,\mathtt{P}\rangle} f_{\mathcal{L}} \mid\ \sigma \in \mathbb{M}\ \}$, where $\mathbb{M}$ is the set of memories (see Section 2.5 for the formal details).

Let us recall that the aim of an obfuscator is to modify a program in order to make it more difficult to analyse while preserving its functionality [CTL98]. In [JGM12], the authors interpret these features, specifying when a program transformation is an obfuscator, in the semantic setting. Following this view, we obtain the following characterisation of an obfuscator:

- An obfuscation transformer $\mathfrak{O}$ has to preserve the denotational semantics of programs, namely the denotational semantics of a program $\mathtt{P}$ and of its obfuscated version $\mathfrak{O}(\mathtt{P})$ have to be the same, i.e., $[\![\mathtt{P}]\!] = [\![\mathfrak{O}(\mathtt{P})]\!]$.
- An obfuscator has to add confusion with respect to some properties that are revealed by the non-obfuscated program $\mathtt{P}$, thus generating an obfuscated program $\mathfrak{O}(\mathtt{P})$ from which the same properties cannot be precisely extracted. Let $\rho, \eta \in \mathit{uco}(\Sigma)$ and assume that the pair of abstractions $\langle\rho, \eta\rangle$ is $\mathscr{B}$-complete for $[\![\mathtt{P}]\!]$. This means that this pair of properties can be precisely extracted form the non-obfuscated program $\mathtt{P}$, namely $\rho([\![\mathtt{P}]\!]) = [\![\mathtt{P}]\!]^{\langle\rho,\eta\rangle} \overset{\text{def}}{=} \{\ \mathit{lfp}_{\langle\sigma,\mathtt{P}\rangle} \rho \circ f_{\mathcal{L}} \circ \eta \mid\ s \in \Sigma\ \}$. Obfuscator $\mathfrak{O}$ obfuscates

$\langle \rho, \eta \rangle$ when $[\![\mathtt{P}]\!]^{\langle \rho,\eta \rangle} \sqsubset [\![\mathfrak{O}(\mathtt{P})]\!]^{\langle \rho,\eta \rangle}$, which holds if and only if $\rho([\![\mathfrak{O}(\mathtt{P})]\!]) \sqsubset [\![\mathfrak{O}(\mathtt{P})]\!]^{\langle \rho,\eta \rangle}$, being $[\![\mathtt{P}]\!]^{\langle \rho,\eta \rangle} = \rho([\![\mathtt{P}]\!]) = \rho([\![\mathfrak{O}(\mathtt{P})]\!])$ [Gia08]. In other words, a property is obfuscated if and only if it is incomplete for the obfuscated program.

In the following, in Section 4.1 we present our attack model, next in Section 4.2 we describe how we can characterise the potency range of an obfuscator thanks to the incompleteness results of Section 3 and then we conclude by providing examples of how this charaterization works in the case of obfuscations that aim at obstructing program slicing (Section 4.3) and static disassembly (Section 4.4).

4.1. Attack Model

Automatic reverse-engineering techniques typically consist in static program analysis (e.g., data flow analysis, control flow analysis, alias analysis, program slicing) and dynamic program analysis (e.g., dynamic testing, profiling, program tracing). Hence, we have two kinds of attacks: one that executes the program, collects computational traces, and then analyses these traces looking for invariants, and the other that statically analyses the code. In other words, dynamic attacks can extract properties of the execution traces, for instance by using data mining techniques, while static attacks analyse the code looking for dynamic properties without executing the program. It is well known [CC77, CC79b] that static analysis can be perfectly modelled in the context of abstract interpretation, where a property is extensionally represented as the set of all the data satisfying it and describes the abstraction of the corresponding data. In particular, static analysis is performed as an abstract execution of programs, namely as the (fix-point) semantic computation on the approximated/abstract data expressing the property of interest. Instead, dynamic analysis can be modelled as an approximated observation of the concrete execution since it describes partial knowledge of the real execution. In the following, we model a property as the function η mapping data to the minimal property containing it. This implies that η is extensive (i.e., $X \subseteq \eta(X)$), namely it approximates by adding noise, it is idempotent since the whole approximation is added in one shot and finally it is monotone, preserving the approximation order. Namely, it is an upper closure operator and the framework beneath is abstract interpretation [CC77, CC79b]. As seen in the previous sections the set $\wp(\Sigma^*)$ can be used to represent the set of possible program semantics, where Σ denotes the set of possible program states. Thus, we view attacks, i.e., static program analysers, as properties of program states that model the abstract domain of computation of program semantics, namely as an analysis over the program semantics.

4.2. Modelling the obfuscation potency range

In [Gia08] and in [JGM12] the objective is to provide an incompleteness-driven *construction* of a potent obfuscator, while here we aim to use this incompleteness-based characterisation for "measuring" potency. In fact, we aim at defining formal domain transformers inducing incompleteness that allow us to systematically characterise a range of analyses that are made incomplete, and therefore imprecise, by the performed code obfuscation. In particular, if $\mathcal{A}$ is a closure operator that models an attacker that can succeed in extracting the desired information, then the incomplete compression $\mathcal{UR}(\mathcal{A})$ (defined in the previous section) characterises the most abstract domain such that any abstract analysis between $\mathcal{A}$ (excluded) and $\mathcal{UR}(\mathcal{A})$ (included) is obfuscated. We describe in details how to extract the range of attackers that are defeated by an obfuscation with respect to a given semantics. We consider the following general scenario where:

- $\mathcal{S} \overset{\text{def}}{=} \mathbb{P}_{\mathcal{L}} \to \mathcal{D}$ is a program (semantic) observation on the set of denotations $\mathcal{D}$. For instance in [JGM12] we consider $\mathcal{S} = [\![\cdot]\!]_{\text{CFG}}$ as the function modelling programs as control flow graphs;
- $\mathfrak{O} : \mathit{Progr} \to \mathit{Progr}$ is an obfuscation transformer, designed for deceiving observations of $\mathcal{S}(\mathtt{P})$, for instance when dealing with CFG we consider obfuscations obscuring the control structure of a program [JGM12];
- $\mathcal{A}$ is the analysis the attacker may perform on the given model, i.e., an abstraction of $\mathcal{S}(\mathtt{P})$.

Our goal is to formally characterise the potency range of the given obfuscator $\mathfrak{O}$ on the considered model $\mathcal{S}$, namely we aim at characterising the most precise analysis, namely attacker, $\mathcal{A}$ unable to disclose a precise information from the abstraction $\mathcal{S}$ of the obfuscated program. In order to characterise the potency of an obfuscation technique, we consider the completeness equation given before, instantiated to the pair of abstractions $\langle \mathit{id}, \mathcal{A} \rangle$, meaning that, when completeness holds, the analysis $\mathcal{A}$ is able to disclose precisely the same information revealed by the considered (concrete) semantics. In the following, we have to consider two different equations of completeness, depending on the way $\mathcal{S}$ is computed. If the abstract program semantics of a program $\overline{\mathtt{P}} = \mathfrak{O}(\mathtt{P})$ is obtained simply as a function of the program, then

$$\forall \overline{\mathtt{P}} \in \mathfrak{O}(\mathbb{P}_{\mathcal{L}}).\ \mathcal{S}(\overline{\mathtt{P}}) = \mathcal{S} \circ \mathcal{A}(\overline{\mathtt{P}})$$

means precisely that, the attacker $\mathcal{A}$ wins, being able to observe of the semantics of the obfuscated program $\overline{\mathtt{P}}$ exactly what the program semantics itself makes available, at least on the considered model.
While, if $\mathcal{S}$ is computed as fix-point of a (semantic) operator $\varphi_{\mathcal{L}}$, inductively defined on the language structure and on the set of states $\Sigma_{\mathcal{S}}$, i.e., $\mathcal{S} \overset{\text{def}}{=} \lambda \mathtt{P} \in \mathbb{P}_{\mathcal{L}}.\ \left\{\ \mathit{lfp}_{\langle \sigma, \mathtt{P}\rangle} \varphi_{\mathcal{L}} \mid \sigma \in \Sigma_{\mathcal{S}}\ \right\}$, then the attacker analysis has to be included in the fix-point computation, namely

$$\forall \overline{\mathtt{P}} \in \mathbb{P}_{\mathcal{L}}.\ \left\{\ \mathit{lfp}_{\langle \sigma, \overline{\mathtt{P}}\rangle} \varphi_{\mathcal{L}} \mid \sigma \in \Sigma_{\mathcal{S}}\ \right\} = \left\{\ \mathit{lfp}_{\langle \sigma, \overline{\mathtt{P}}\rangle} \varphi_{\mathcal{L}} \circ \mathcal{A} \mid \sigma \in \Sigma_{\mathcal{S}}\ \right\}$$

In this case, the meaning is the same as before, only the way the semantics are computed is changed.
This completeness equation, allows to characterise the *potency range* for the obfuscator $\mathfrak{O}$ since all the attackers between the most concrete complete shell [GRS00] (excluded) and the most abstract incomplete compression (Section 3) are attackers against which the obfuscator is potent. In our case, the input observation is the identity and therefore also the completeness shell is the identity, and the semantic function is $f_{\mathcal{S}} = \varphi_{\mathcal{L}}$ (in the case where $\mathcal{S}$ is computed inductively on the language structure) or $f_{\mathcal{S}} = \mathcal{S}$ (otherwise). Hence, the potency range of $\mathfrak{O}$ is modelled as the set of the analyses defeated by $\mathfrak{O}$, i.e.,

$$\mathcal{P}\mathrm{ot}_{\mathfrak{O},\mathcal{S}} \overset{\text{def}}{=} \left\{\ \mathcal{A} \,\middle|\, \mathcal{A} \sqsubseteq \mathcal{U}\mathcal{R}^{\mathscr{B}}_{f_{\mathcal{S}},id}(id)\ \right\}$$

It is clear that, if we have a starting attack analysis $\overline{\mathcal{A}}$ on which focusing the characterisation, then the same set can be computed parametrically on $\overline{\mathcal{A}}$:

$$\mathcal{P}\mathrm{ot}^{\overline{\mathcal{A}}}_{\mathfrak{O},\mathcal{S}} \overset{\text{def}}{=} \left\{\ \mathcal{A} \,\middle|\, \mathcal{R}^{\mathscr{B}}_{f_{\mathcal{S}},id}(\overline{\mathcal{A}}) \sqsubset \mathcal{A} \sqsubseteq \mathcal{U}\mathcal{R}^{\mathscr{B}}_{f_{\mathcal{S}},id}(\overline{\mathcal{A}})\ \right\}$$

This not the first attempt to model potency by means of abstract interpretation. In [DG05], the basic idea is to define potency in terms of the most concrete *output* observation left unchanged by the obfuscation, i.e., $\delta_{\mathfrak{O}}$ such that $\delta_{\mathfrak{O}}(\llbracket \mathtt{P} \rrbracket) = \delta_{\mathfrak{O}}(\llbracket \mathfrak{O}(\mathtt{P}) \rrbracket)$. The set of all the obfuscated properties, making the obfuscator potent, is determined by all the analyses $\{\ \mathcal{A} \mid \mathcal{A}$ not more abstract than $\delta_{\mathfrak{O}}\ \}$.
At this point, we compute the incomplete compressor for this equation in order to characterise the most concrete abstraction $\mathcal{A}$ making completeness fail, namely the maximal (most abstract) observation for which the obfuscator is *potent*. In fact, incompleteness means that the attacker $\mathcal{A}$ on the obfuscated program discloses an imprecise approximation of the information concerning the program semantics. The following example uses the proposed approach in order to identify the range of potency of a data obfuscation techniques with respect to the computation of the square function.

Example 4.1. Let us consider data obfuscation, and in particular the incompleteness characterisation provided in [JGM12]. This obfuscation technique is based on the encoding of data [DTM07]. In this case obfuscation is achieved by data-refinement, namely by exploiting the complexity of more complex data-structures or values in such a way that actual computations can be viewed as abstractions of the refined (obfuscated) ones.

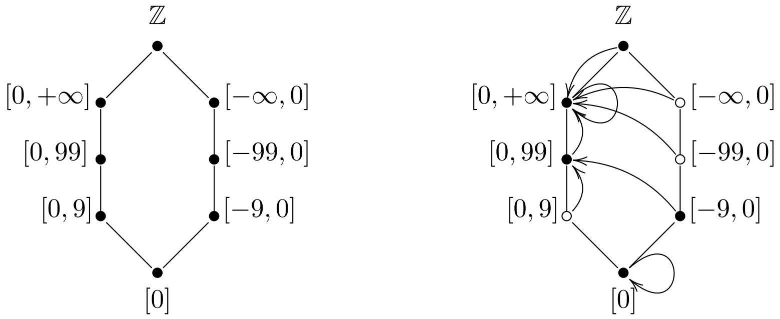
The idea consists in choosing a pair of statements c^{α} and c^{γ} such that $c^{\gamma}; c^{\alpha} \equiv \mathbf{skip}$. This means that both c^{α} and c^{γ} are statements of the form:

$$c^{\alpha}\ \equiv\ x := G(x) \qquad \text{and} \qquad c^{\gamma}\ \equiv\ x := F(x),$$

for some function F and G. A program transformation $\mathfrak{O}(\mathtt{P}) \overset{\text{def}}{=} c^{\gamma}; \tau_x(\mathtt{P}); c^{\alpha}$ is a data-type obfuscation for data-type x if $\mathfrak{O}(\mathtt{P}) \equiv \mathtt{P}$, where τ_x adjusts the data-type computation for x on the refined type (see [DTM07]).

It is known that data-type obfuscation can be modelled as adjoint functions (Galois connections), where c^{γ} represents the program concretizing, viz. refining, the datum x and c^{α} represents the program abstracting the refined datum x back to the original data-type. As proved in [Gia08], this is precisely modelled as a pair of adjoint functions: $\alpha : \mathit{Val} \longrightarrow \mathit{Val}^{\Re}$ and $\gamma : \mathit{Val}^{\Re} \longrightarrow \mathit{Val}$ relating the standard data-type Val for x with its refined version $\mathit{Val}^{\Re}$. For instance, consider

$$\mathtt{P} =\ x := x + 2;, \qquad c^{\alpha} \equiv x := x/2 \qquad \text{and} \qquad c^{\gamma} \equiv x := 2x,$$



$\text{I} = \rho_{\text{I}}(\wp(\mathbb{Z}))$ (a)

Fig. 4. The abstract domain I and its abstraction ρ_a

then we have $\tau_x(\mathtt{P}) =\ x := 2(x/2+2)$, namely $x := x+4$, therefore:

$\mathfrak{O}(\mathtt{P}) \equiv x := 2x; x := x+4; x := x/2.$

Consider now a slightly more complex example, for instance the program:

$$\mathtt{P} = \left[\begin{array}{l} x := 1; s := 0; \\ \textbf{while } x < 15 \textbf{ do } s := s + x; x := x + 1; \textbf{ endw} \end{array}\right.$$

Then

$$\tau_x(\mathtt{P}) = \left[\begin{array}{l} x := 2; s := 0; \\ \textbf{while } x < 30 \textbf{ do } s := s + x/2; x := x + 2; \textbf{ endw} \end{array}\right.$$

where α, γ, *Val*, and $\mathit{Val}^{\Re}$ are the most obvious ones. In [JGM12], given $\rho \in \mathit{uco}(\mathit{Val})$, we showed that this obfuscation can be modelled as a distorted self interpreter adding dummy uses of a specific syntactic operation *op*, for whose semantics the abstraction ρ is incomplete. Let *op* such a syntactic operation of the language $\mathcal{L}$. Then we can compute the maximal (most abstract) incomplete observation for this particular operator w.r.t. the completeness equation

$[\![op]\!] = [\![op]\!] \circ \rho$

where $\mathcal{S} = [\![\cdot]\!]$ and $\mathcal{A} = \rho$. Again, in this case we look for the most concrete observation unable to disclose precisely the information released, in this case, by the concrete semantics $[\![\cdot]\!]$. Let I be the domain in Figure 4 and $op_{\mathtt{P}} = sq^* \overset{\text{def}}{=} \rho_{\text{I}} \circ sq$, then $\rho_a = \{\mathbb{Z}, [0,+\infty], [0,99], [-9,0], [0]\}$ (black dots in Figure 4 (a)) is not $\mathscr{B}$-complete on the concrete domain I for sq^*. Then the refined domain

$\rho' \overset{\text{def}}{=} \mathcal{R}^{\mathscr{B}}_{[\![op_{\mathtt{P}}]\!]}(\rho_a) = \{\mathbb{Z}, [0,+\infty], [0,99], [0,9], [-9,0], [0]\}.$

is complete for $op_{\mathtt{P}}$ by construction, hence it can be used for characterising the potency of the obfuscation technique consisting in using the operator $op_{\mathtt{P}}$ in a program.

At this point, we can compute

$\mathcal{UR}(\rho') = \mathcal{UR}(\rho_a) = \mathcal{M}(\mathit{Mirr}(\rho' \sqcap R_{[\![sq^*]\!]}(\rho')) \smallsetminus R_{[\![sq^*]\!]}(\rho')) = \mathcal{M}(\mathit{Mirr}(\rho') \smallsetminus R_{[\![sq^*]\!]}(\rho')).$

where $R_{[\![sq^*]\!]}(\rho') = \{\mathbb{Z}, [0,9], [-9,0], [0]\}$. Hence, the resulting incomplete compression is the domain $\mathcal{UR}(\rho') = \{\mathbb{Z}, [0,+\infty], [0,99]\}$. In this way, we provide a model of the potency of the obfuscation technique since we know that all the analyses between ρ' (excluded) and $\mathcal{UR}(\rho')$ (included) are made imprecise by the performed code transformation.

4.3. Obfuscating Program Slicing

In this section, we describe slicing [HRB90, Wei81] as an abstraction of a program semantics constructing the program dependency graph (PDG for short). In particular, we show that slicing obfuscation [MDT07] against attackers performing slicing analyses, is potent when there are syntactic dependencies between variables that do not correspond to semantic dependencies. For instance, in the assignment $y = x + 1$ there is a semantic dependency of y on x, while in $y = x + 5 - x$ there is a syntactic dependency between y and x, such that the value of y does not depend on x. Let us provide a brief overview on program slicing [Wei81] and on the way slices are computed in [HRB90].

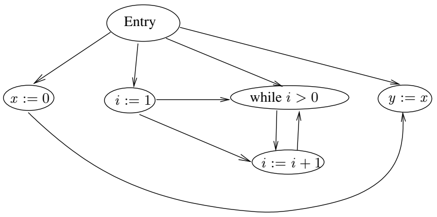
Definition 4.2 ((Semantic) Program slicing). For a variable v and a statement (program point) s (final use of v), the slice S of program $\mathtt{P}$ with respect to the slicing criterion $\langle s, v\rangle$ is any executable program such that S can be obtained by deleting zero or more statements from $\mathtt{P}$ and if $\mathtt{P}$ halts on input I then the value of v at the statement s, each time s is reached in $\mathtt{P}$, is the same in $\mathtt{P}$ and in S. If $\mathtt{P}$ fails to terminate then s may be reached more times in S than in $\mathtt{P}$, but $\mathtt{P}$ and S execute the same value for v each time s is executed by $\mathtt{P}$.

The standard approach for characterising slices is based on PDG [HRB90]. A *program dependence graph* [GL91] $\mathcal{P}_\mathtt{P}$ for a program $\mathtt{P}$ is a directed graph with vertexes denoting program components and edges denoting *dependencies* between components. The vertexes of $\mathcal{P}_\mathtt{P}$, $\mathit{Nodes}(\mathcal{P}_\mathtt{P})$, represent the assignment statements and control predicates that occur in $\mathtt{P}$. In addition $\mathit{Nodes}(\mathcal{P}_\mathtt{P})$ includes a distinguished vertex called *Entry* denoting the starting vertex. An edge represents either a *control dependence* or a *flow dependence*. Control dependence edges $u \longrightarrow_c v$ are such that (1) u is the *Entry* vertex and v represents a component of $\mathtt{P}$ that is not nested within any control predicate; or (2) u represents a control predicate and v represents a component of $\mathtt{P}$ immediately nested within the control predicate represented by u. Flow dependence edges $u \longrightarrow_f v$ are such that (1) u is a vertex that defines variable x (an assignment), (2) v is a vertex that uses x, and (3) Control can reach v after u via an execution path along which there is no intervening definition of x. Finally, on these graphs, a slice for a criterion $\langle s, v\rangle$ is the sub-graph containing all the vertexes that can reach s via flow/control edges. It is worth noting that these slices are characterised by means of syntax-based dependencies, therefore in general they are not the smallest program fragments satisfying Definition 4.2 [MZ08].

Example 4.3. Consider the following programs [RT96] and note that $\mathtt{P}_2$ is a slice of $\mathtt{P}_1$.

$$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\mathbf{while}\ i > 0\ \mathbf{do}\ i := i + 1\,; \\ {}^{4.}y := x\,; \end{array}\right. \qquad \mathtt{P}_2 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ \\ {}^{4.}y := x\,; \end{array}\right.$$

Below, we find a representation of the program dependence graph of $\mathtt{P}_1$. In this representation, we have only control and flow dependence edges, without distinction. In this graph we can note that slice $\mathtt{P}_2$ (with criterion the value of y) can be computed by following backwards the edges starting from node $y := x$, the final definition of y.



Before defining formally the construction of the program dependency graph, we can observe that the notion of slicing given in Definition 4.2 is strongly based on a notion of dependency between statements, and therefore between variables of the program [MZ08]. For this reason, we first need to formalize the notion of dependence that we use in the following:

Definition 4.4 (Semantic dependencies). [MZ08][5] Let $x \in \mathit{Var}$, $Y \subseteq \mathit{Var}$. We say that an expression $\mathtt{e}$ depends on x, written $x \in \mathit{Dep}(\mathtt{e})$, if $\exists \sigma_1.\sigma_2 \in \mathbb{M}.\ \forall y \neq x.\ \sigma_1(y) = \sigma_2(y)\ \wedge\ [\![\mathtt{e}]\!]\sigma_1 \neq [\![\mathtt{e}]\!]\sigma_2$.

In the following, we extend this notion directly to the statement containing an expression. In particular, we say that a statement $\mathtt{c}$ depends on $Y \subseteq \mathit{Var}$, written $Y \subseteq \mathit{Dep}(\mathtt{c})$, if for each $y \in Y$ there exists an expression $\mathtt{e}$ in the statemet $\mathtt{c}$ such that $y \in \mathit{Dep}(\mathtt{e})$.

[5] In [MZ08], this notion of dependency can be tuned depending on the degree of precision we need in the characterisation of the slicing, moving towards *abstract slicing*.

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

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

Fig. 5. Original and obfuscated programs.

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

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

Fig. 6. Slices of the original program.

Program slicing obfuscation: Introducing fake dependencies.

Program slicing obfuscation consists in a program obfuscation deceiving the program slicing transformation technique [MDT07]. Since, as we have seen above, program slicing is based on the notion of dependency between variables and expressions, intuitively it is clear that, if we aim at deceiving this analysis technique, we have to introduce in the program useless (syntactic) dependencies that do not affect the semantics, we call these dependencies *fake dependencies.* A fake dependence is precisely a syntactic dependence between variables that do not correspond to semantic dependence. The introduction of such dependencies could be realized, for instance, by modifying an assignments $x := e$ by using some irrelevant variable z in the following way: $x := (z + e) - z$. Its effect would be that a naive program flow analyser would think that all assigned values depended on variable z.

Example 4.5. In order to explain the idea of slicing obfuscation, let us consider the word count program [MDT07] given in Figure 5 (where **getchar()** returns the next char in a file, while **out** is the procedure output instruction), on the left. It takes in a block of text and outputs the number of lines (nl), words (nw) and characters (nc). Suppose the slicing criterion is $(nl, 8)$, then the slice is on the left in Figure 6, while if the criterion is $(nw, 8)$ then the slice is on the right.

Let us introduce some fake dependencies. In particular, we modify line 7 adding an opaque predicate which is always true and we add lines 8 and 9 with opaque predicates that are always false. The obfuscated program is given in Figure 5, on the right.

In this way semantically we do not change the dependencies but a syntactic analysis would observe that *nl*, *nw* and *nc* depend one on the others. Hence, the new slices are in Figure 7.

Semantic PDG as abstraction of program semantics.

In this section, we characterise the PDG construction as an abstract interpretation of a program. In other words, we define the abstract interpreter whose fix point computes the program PDG. Let us first define a semantic function, similar to graph semantics [JGM12] which, instead of computing the control flow graph, computes the PDG obtained by including only semantic dependencies among variables.
We consider here an approximation of PDG given in the literature [RY89] as we will show later. A PDG $\mathcal{P}_l$ is defined as a pair $\langle Nodes(\mathcal{P}_l), Arcs(\mathcal{P}_l)\rangle$. In particular, we have two kinds of edges, the *control* dependence edges $Control(\mathcal{P}_l)$ and the *flow* dependence edges $Flow(\mathcal{P}_l)$, hence $Arcs(\mathcal{P}_l) = Control(\mathcal{P}_l) \cup Flow(\mathcal{P}_l)$. Let

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

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

Fig. 7. Slices of the obfuscated program.

us denote by *PDG* the set of program dependency graphs, ordered by set inclusion on the sets of nodes and arcs. Formally, $PDG = (\wp_f(\mathbb{N}) \times \wp_f(\mathbb{N} \times \mathbb{N}), \leq_{PDG})^6$, where $\mathcal{P} \leq_{PDG} \mathcal{P}'$ if and only if $\mathit{Nodes}(\mathcal{P}) \subseteq \mathit{Nodes}(\mathcal{P}')$ and $\mathit{Arcs}(\mathcal{P}) \subseteq \mathit{Arcs}(\mathcal{P}')$. It is worth noting that *PDG* is not a complete lattice (see also [Rep91]) since we can find an infinite ascending chain whose union is an infinite set, not belonging to the domain, for the same reason this domain has no meet irreducible elements. Hence, in order to have a complete lattice, we need to fix the program of interest, and indeed only referring to a particular program we can characterise the potency of slicing obfuscation on the program. Let us define the *PDG* $\mathcal{P}_{\mathtt{P}}$ in the following way

$$\mathit{Nodes}_{\mathtt{P}} \overset{\text{def}}{=} \{\ l \in \mathbb{N} \mid \ \mathtt{Stm}_{\mathtt{P}}(l) \in \mathtt{P}\ \ \} \subseteq \mathbb{N} \text{ as the set of all the locations of } \mathtt{P}$$
$$\mathit{Control}_{\mathtt{P}} \overset{\text{def}}{=} \{\ \langle l_1, l_2\rangle \mid \ \mathtt{Stm}(l_1) \in \{\textbf{if}, \textbf{while}\} \text{ and } \mathtt{Stm}(l_2) \text{ is nested in } l_1\ \ \}$$
$$\mathit{Flow}_{\mathtt{P}} \overset{\text{def}}{=} \mathit{Nodes}_{\mathtt{P}} \times \mathit{Nodes}_{\mathtt{P}}$$

where we recall that $\mathtt{Stm}_{\mathtt{P}} : \mathbb{N} \longrightarrow \mathbb{P}$ is the function s.t. $\mathtt{Stm}_{\mathtt{P}}(l)$ is the statement in program point l. At this point, we define the complete lattice $\langle \mathit{PDG}_{\mathtt{P}}, \leq_{PDG}, \mathcal{P}_{\mathtt{P}}, \varnothing, \vee_{PDG}, \wedge_{PDG}\rangle$ parametric on the program $\mathtt{P}$, where $\mathit{PDG}_{\mathtt{P}} \overset{\text{def}}{=} \{\ \mathcal{P} \in \mathit{PDG} \mid \ \mathit{Nodes}(\mathcal{P}) \subseteq \mathit{Nodes}_{\mathtt{P}}, \mathit{Control}(\mathcal{P}) \subseteq \mathit{Control}_{\mathtt{P}}, \mathit{Flow}(\mathcal{P}) \subseteq \mathit{Flow}_{\mathtt{P}}\ \ \}$. We can observe that the top of this lattice is precisely the PDG of the program $\mathtt{P}$ where all the possible dependencies are considered in the flow edges. In other words, they are all the PDGs where all the possible dependencies, but one, are included in the graph, or the PDGs missing the last executable node.

At this point, we define the program semantics generating the PDG of a program while interpreting the program. Given a program $\mathtt{P}$, its semantics is defined by means of a transition system $\langle \Sigma_{\mathtt{P}}, \mathtt{p}\rangle$ with $\Sigma_{\mathtt{P}} = \Sigma \cup \{\mathtt{P}\}$ and transition function $\mathtt{p}$. The states have the form $\langle \sigma, \langle l, l'\rangle, \mathcal{P}_l, D_l\rangle \in \Sigma$, where $\sigma \in \mathbb{M}$ is the memory, namely the actual values of program variables, $\langle l, l'\rangle \in \mathbb{N} \times \mathbb{N}$ is a pair of program points, l is the executed statement in $\mathtt{P}$ and l' is the next statement to execute in $\mathtt{P}$, $\mathcal{P}_l \in \mathit{PDG}_{\mathtt{P}}$ is the PDG of $\mathtt{P}$ computed upto program point l, and D_l is a definition function associating, at the program point l, with each variable, the program point where the variable has been defined

$$D_l : \mathit{Var} \longrightarrow \mathbb{N} \ \text{ s.t. } D_l(x) = n \text{ where } n \text{ is the program point where } x \text{ is defined}$$

For instance, in Example 4.3, $D_4(x) = 1$ while $D_4(y) = 4$. Note that, the characterisation of D_l may be ambiguous or imprecise in the case when we have more than one definition of the same variable in the program. In general, we can avoid this ambiguity by supposing the programs translated in their single static assignment (SSA) form [CFR$^+$91], where each variable is defined precisely once, in one single program point.

Finally, we have to define the transition function generating dependency-based PDG of $\mathtt{P}$. By dependency-based we mean that the flow edges will be generated by considering semantic dependencies, the ones defined in Definition 4.4. First of all, for each program point $l \in \mathbb{N}$ we have to define the following auxiliary maps: $\mathtt{Dep}(l)$ denotes the set of variables the statement in l depends on, while $\mathtt{Use}(l)$ denotes the set of variables used in l, and it is defined by structural induction in Table 2.

[6] By $\wp_f(X)$ we denote the domain of all the *finite* subsets of X.

$\mathtt{Use}(\mathbf{skip}) = \varnothing$	$\mathtt{Use}(x := \mathtt{e}) = \mathit{Var}(\mathtt{e})$	$\mathtt{Use}(\mathtt{C}_1 ; \mathtt{C}_2) = \mathtt{Use}(\mathtt{C}_1) \cup \mathtt{Use}(\mathtt{C}\,2)$
$\mathtt{Use}(\mathbf{while}\ \mathtt{e}\ \mathbf{do}\ \mathtt{C}\ \mathbf{endw}) = \mathit{Var}(\mathtt{e}) \cup \mathtt{Use}(\mathtt{C})$	$\mathtt{Use}(\mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{C}_0\ \mathbf{else}\ \mathtt{C}_1\ \mathbf{fi}) = \mathit{Var}(\mathtt{e}) \cup \mathtt{Use}(\mathtt{C}_1) \cup \mathtt{Use}(\mathtt{C}_2)$	

Table 2. The inductive definition of `Use`

$$\mathtt{Dep} : \mathbb{N} \longrightarrow \wp(\mathit{Var}) \text{ s.t. } \mathtt{Dep}(l) = \bigcup_{\mathtt{e} \in \mathtt{Stm}_{\mathtt{P}}(l)} \mathit{Dep}(\mathtt{e}) \qquad \text{and}$$

$$\mathtt{Use} : \mathbb{N} \longrightarrow \wp(\mathit{Var}) \text{ s.t. } \mathtt{Use}(l) = \bigcup_{\mathtt{e} \in \mathtt{Stm}_{\mathtt{P}}(l)} \mathit{Var}(\mathtt{e})$$

where $\mathtt{e} \in \mathtt{Stm}_{\mathtt{P}}(l)$ means that the expression $\mathtt{e}$ is syntactically present in the statement $\mathtt{Stm}_{\mathtt{P}}(l)$, and $\mathit{Dep}(\mathtt{e})$ is defined in Definition 4.4. In general, $\mathtt{Dep}(l) \subseteq \mathtt{Use}(l)$, if there are no fake dependencies we have the equality. Another necessary function for characterising the history of computation, is the function determining the sequence of program points executed: $\mathtt{Next}_{\mathtt{P}} : \mathbb{M} \times \mathbb{PL}_{\mathtt{P}} \rightarrow \mathbb{PL}_{\mathtt{P}}$

$$\begin{array}{lll} \mathtt{Next}_{\mathtt{P}}(\sigma, l) = l' & \text{iff} & f_{\mathcal{L}}(\langle \sigma, \mathtt{Stm}_{\mathtt{P}}(l) \rangle) = \langle \sigma', \mathtt{C} \rangle \ \wedge \ \mathtt{Pc}_{\mathtt{P}}(\mathtt{C}) = l' \\ \mathtt{Next}_{\mathtt{P}}(\sigma, l) = l & \text{if} & l \text{ is the last statement of } \mathtt{P} \qquad \text{(In this way we can reach a fix-point.)} \end{array}$$

The dependency-based transition function is $\mathtt{p}(\langle \sigma, \langle l_1, l_2 \rangle, \mathcal{P}_{l_1}, D_{l_1} \rangle) = \langle \sigma', \langle l_2, \mathtt{Next}_{\mathtt{P}}(\sigma', l_2) \rangle, \mathcal{P}_{l_2}, D_{l_2} \rangle$ where σ' is the memory modified by executing statement in l_1, $\mathtt{Next}_{\mathtt{P}}$ computes the following statement to execute, $D_{l_2} = D_{l_1}[D_x = l_2]$ if $\mathtt{Stm}(l_2) = x := \mathtt{e}$ (it is D_{l_1} otherwise), and $\mathcal{P}_{l_2}$ is computed as follows:

$$\begin{array}{rl} \mathit{Nodes}(\mathcal{P}_{l_2}) & = \mathit{Nodes}(\mathcal{P}_{l_1}) \cup \{l_2\} \cup \{\ l \mid \mathtt{Stm}(l_2) \in \{\mathbf{if}, \mathbf{while}\} \text{ and } \mathtt{Stm}(l) \text{ is nested in } l_2\ \} \\ \mathit{Control}(\mathcal{P}_{l_2}) & = \mathit{Control}(\mathcal{P}_{l_1}) \cup \{\ \langle l_2, l \rangle \mid \mathtt{Stm}(l_2) \in \{\mathbf{if}, \mathbf{while}\} \text{ and } \mathtt{Stm}(l) \text{ is nested in } l_2\ \} \\ \mathit{Flow}(\mathcal{P}_{l_2}) & = \mathit{Flow}(\mathcal{P}_{l_1}) \cup \{\ \langle D_{l_1}(x), l_2 \rangle \mid x \in \mathtt{Dep}(l_2)\ \} \end{array}$$

In other words, $\langle n, l' \rangle \in \mathit{Flow}(\mathcal{P}_l)$ if there exists $x \in \mathit{Var}$ such that $x \in \mathtt{Dep}(l') \cap \{\ x \mid \mathtt{Stm}(n) = x := \mathtt{e}\ \}$. It is worth noting that, if we reach the end of the program then $\mathtt{Next}_{\mathtt{P}}(\sigma, l) = l$, and consequently also $\mathtt{p}$ reaches the fix point.

The PDG operational semantics is $[\![\mathtt{P}]\!]_{\mathit{PDG}} \overset{\text{def}}{=} \{\ \mathit{lfp}_{\langle \sigma, \mathtt{P} \rangle}\, \mathtt{p} \mid \sigma \in \Sigma\ \}$. Intuitively, by using this transition function we compute the PDG with flow edges considering only those variables an expression *depends on*. In fact, $\mathit{Flow}(\mathcal{P}_{l_2})$ is obtained by adding to $\mathit{Flow}(\mathcal{P}_{l_1})$ the flows from the variables in $\mathtt{Dep}(l_2)$. In general, this is more precise than the standard construction of the PDG, which is provided in terms of the variables *used* in an expression. This suggests us that we can generalize this construction in terms of an *abstraction* of states given in terms of abstraction of the flow edges.

In order to be as general as possible, consider a function $\mathtt{F} : \mathbb{N} \longrightarrow \wp(\mathit{Var})$ such that $\mathtt{Dep}(l) \subseteq \mathtt{F}(l)$, that approximates the variables a program point depends on, and the corresponding abstractions $\alpha_s^{\mathtt{F}} : \Sigma_{\mathtt{P}} \longrightarrow \wp(\Sigma_{\mathtt{P}})$ (denoting also its additive lift), $\alpha_p^{\mathtt{F}} : \mathit{PDG}_{\mathtt{P}} \longrightarrow \mathit{PDG}_{\mathtt{P}}$, and $\alpha_f^{\mathtt{F}} : \wp(\mathit{Flow}_{\mathtt{P}}) \longrightarrow \wp(\mathit{Flow}_{\mathtt{P}})$ defined as follows:

$$\alpha_s^{\mathtt{F}}(\langle \sigma, \langle l, l' \rangle, \mathcal{P}_l, D_l \rangle) \overset{\text{def}}{=} \langle \sigma, \langle l, l' \rangle, \alpha_p^{\mathtt{F}}(\mathcal{P}_l), D_l \rangle$$
$$\alpha_p^{\mathtt{F}} \in \mathit{uco}(\mathit{PDG}_{\mathtt{P}}),\ \alpha_p^{\mathtt{F}}(\mathcal{P}_l) = \mathcal{P}'_l \qquad \text{such that} \qquad \mathit{Nodes}(\mathcal{P}'_l) = \mathit{Nodes}(\mathcal{P}_l) \text{ and}$$
$$\mathit{Control}(\mathcal{P}'_l) = \mathit{Control}(\mathcal{P}_l) \text{ and } \mathit{Flow}(\mathcal{P}'_l) = \alpha_f^{\mathtt{F}}(\mathit{Flow}(\mathcal{P}_l)) \overset{\text{def}}{=} \{\ \langle D_l(x), l' \rangle \mid l' \in \mathit{Nodes}(\mathcal{P}_l),\ x \in \mathtt{F}(l')\ \}$$

where $\alpha_f^{\mathtt{F}} \in \mathit{uco}(\wp(\mathit{Flow}_{\mathtt{P}}))$. It is clear that $\alpha_s^{\mathtt{F}}$ induces an abstraction on states in the sense that it considers more abstract PDG in the domain $\mathit{PDG}_{\mathtt{P}}$, since we consider only $\mathtt{F}$ such that $\forall l \in \mathbb{N}.\, \mathtt{F}(l) \supseteq \mathtt{Dep}(l)$, namely $\mathtt{F}$ approximating the dependency function used for computing $\mathcal{P}_l$. For instance, if $\mathit{Flow}(\mathcal{P}_l)$ is the one above, defined in terms of $\mathtt{Dep}$, and $\mathtt{F} = \mathtt{Use}$, then the resulting function $\alpha_s^{\mathtt{Use}}$ is an abstraction of states in this sense.

Potency of slicing obfuscation

The characterisation of the PDG semantics of a program can be exploited in order to provide a formal characterisation of when a PDG semantics is precise with respect to the slicing analysis and, on the contrary how we can deceive slicing by obfuscating a program. This can be done by considering a $\mathscr{B}$-completeness formulation of the notion of precision, namely we provide a completeness equation characterising when an abstraction $\alpha_s^{\mathtt{F}}$ is precise: Let us define $[\![\mathtt{P}]\!]_{PDG}^{\alpha_s^{\mathtt{F}}} \overset{\text{def}}{=} \left\{\ \mathit{lfp}_{\langle\sigma,\mathtt{P}\rangle}\mathtt{p}\circ\alpha_s^{\mathtt{F}} \mid \sigma\in\Sigma\ \right\}_{|PDG}$, where $\langle\sigma,\langle l,l'\rangle,\mathcal{P}_l,D_l\rangle_{|PDG} = \mathcal{P}_l$, and $[\![\mathtt{P}]\!]_{PDG} \overset{\text{def}}{=} [\![\mathtt{P}]\!]_{PDG}^{id}$, then the abstraction $\alpha_s^{\mathtt{F}}$ is precise with respect to the dependency observation if the following completeness equation holds

$$[\![\mathtt{P}]\!]_{PDG} = [\![\mathtt{P}]\!]_{PDG}^{\alpha_s^{\mathtt{F}}} \tag{1}$$

corresponding to the general approach provided, with $\mathcal{S} = [\![\cdot]\!]_{PDG}$, while $\mathcal{A} = \alpha_s^{\mathtt{F}}$. We use this equation for proving that the syntactic PDG-based computation of slices [HRB90] can be modelled in this framework by considering the abstraction $\mathcal{U} \overset{\text{def}}{=} \alpha_s^{\mathtt{Use}}$, and that it is inherently incomplete in presence of fake dependencies. The following lemma tells us that the transition function $\mathtt{p}$ preserves strict inclusions between PDGs.

Lemma 4.6. Let $\mathcal{P}_1,\mathcal{P}_2 \in \mathit{PDG}_{\mathtt{P}}$ such that $\mathcal{P}_1 <_{PDG} \mathcal{P}_2$, $\sigma\in\mathbb{M}$, $\langle l,l'\rangle\in\mathbb{N}\times\mathbb{N}$ and D_l a definition function for $\mathtt{P}$, then $\mathtt{p}(\langle\sigma,\langle l,l'\rangle,\mathcal{P}_1,D_l\rangle)_{|PDG} < \mathtt{p}(\langle\sigma,\langle l,l'\rangle,\mathcal{P}_2,D_l\rangle)_{|PDG}$.

Proof. Suppose $\mathcal{P}_1 <_{PDG} \mathcal{P}_2$, it means that $\mathit{Nodes}(\mathcal{P}_1)\subset\mathit{Nodes}(\mathcal{P}_2)$, or $\mathit{Control}(\mathcal{P}_1)\subset\mathit{Control}(\mathcal{P}_2)$ or $\mathit{Flow}(\mathcal{P}_1)\subset\mathit{Flow}(\mathcal{P}_2)$. In any case, by definition, we can observe that $\mathtt{p}$ can only enlarge these sets, and since we are precisely in the same program point of $\mathtt{P}$, it modifies the *PDG*s in the same way, therefore preserving the strict inclusion. □

The following lemma tells us that the PDG computed on a portion of a program $\mathtt{P}$, which does not contain fake dependencies, is the same even if we use the transition function abstracted by $\mathcal{U}$.

Lemma 4.7. A program $\mathtt{P}$ has no fake dependencies upto program point l if and only if

$$\mathtt{p}(\langle\sigma,\langle l,l'\rangle,\mathcal{P}_l,D_l\rangle)_{|PDG} = \mathtt{p}\circ\mathcal{U}(\langle\sigma,\langle l,l'\rangle,\mathcal{P}_l,D_l\rangle)_{|PDG}.$$

Proof. Trivial since, without fake dependencies, we can prove inductively on the syntax that, for each $l\in\mathbb{N}$, $\mathtt{Use}(l)=\mathtt{Dep}(l)$, hence the functions $\mathtt{p}$ and $\mathtt{p}\circ\mathcal{U}$ compute precisely the same PDG. □

The following lemma says that, if we compute the PDG by the *abstract transition function* then we obtain the same set of flow edges computed by abstracting the flow edges generated by the transition function $\mathtt{p}$.

Lemma 4.8. Let $\mathcal{P}_l^{\mathtt{Use}}$ be the PDG computed by using the abstract transition function $\mathtt{p}\circ\mathcal{U}$, namely built step by step by using $\mathtt{Use}$, while $\mathcal{P}_l$ is the PDG computed by using $\mathtt{p}$. Then $\mathit{Flow}(\mathcal{P}_l^{\mathtt{Use}}) = \mathit{Flow}(\mathcal{U}(\mathcal{P}_l))$.

Proof. Consider $\langle l_1,l_2\rangle\in\mathit{Flow}(\mathcal{P}_l^{\mathtt{Use}})$, then $\exists x\in\mathtt{Use}(l_2)\cap\{\ x\mid \mathtt{Stm}_{\mathtt{P}}(l_1)=x:=\mathtt{e}\ \}$, but then $l_1 = D_l(x)$, which implies that $\langle l_1,l_2\rangle\in\mathit{Flow}(\mathcal{U}(\mathcal{P}_l))$ by definition.
Let $\langle l_1,l_2\rangle\in\mathit{Flow}(\mathcal{U}(\mathcal{P}_l))$, then $l_2\in\mathit{Nodes}(\mathcal{P}_l)$, $x\in\mathtt{Use}(l_2)$ and $n = D_l(x)$, but this means that $x\in\mathtt{Use}(l_2)\cap\{\ x\mid\mathtt{Stm}_{\mathtt{P}}(l_1)=x:=\mathtt{e}\ \}$, namely $\langle l_1,l_2\rangle\in\mathit{Flow}(\mathcal{P}_l^{\mathtt{Use}})$. □

The following result proves that when a program $\mathtt{P}$ has fake dependencies then the syntactic computation of the PDG is incomplete, namely less precise than the semantic PDG computation.

Proposition 4.9. Let $\mathtt{P}$ a program with fake dependencies, then $[\![\mathtt{P}]\!]_{PDG} < [\![\mathtt{P}]\!]_{PDG}^{\mathcal{U}}$.

Proof. Suppose $\mathtt{P}$ contains its first fake dependency at the program point l, namely $\mathtt{P}=\mathtt{P}';\mathtt{c};\mathtt{P}''$ with $l=\mathbb{PL}_{\mathtt{P}}(\mathtt{c})$, where $\mathtt{P}'$ does not contain fake dependencies. Let l' the last program point of $\mathtt{P}$ executed before l. Then, by Lemma 4.7, given $\sigma\in\mathbb{M}$ computed by $\mathtt{p}$ upto l', we have that $\mathcal{P}_{l'} = \mathcal{P}_{l'}^{\mathtt{Use}}$. At this point, we consider $\mathtt{p}(\langle\sigma,\langle l',l\rangle,\mathcal{P}_{l'},D_{l'}\rangle)$ and $\mathtt{p}\circ\mathcal{U}(\langle\sigma,\langle l',l\rangle,\mathcal{P}_{l'}^{\mathtt{Use}},D_{l'}\rangle)$, and we observe that these computation differ only in the computation of the set of flow edges. In particular, by Lemma 4.8 we have that $\mathit{Flow}(\mathcal{P}_l^{\mathtt{Use}}) = \{\ \langle D_l(x),l\rangle\mid\ x\in\mathtt{Use}(l)\ \}$, but then we have

$$\mathit{Flow}(\mathcal{P}_l) = \{\ \langle D_l(x),l\rangle\mid\ x\in\mathtt{Dep}(l)\ \}\subset\{\ \langle D_l(x),l\rangle\mid\ x\in\mathtt{Use}(l)\ \} = \mathit{Flow}(\mathcal{P}_l^{\mathtt{Use}})$$

since $\mathtt{Dep}(l)\subset\mathtt{Use}(l)$ due to the fake dependency in l. Hence, $\mathcal{P}_l <_{PDG} \mathcal{P}_l^{\mathtt{Use}}$. But this, by Lemma 4.6, implies that also the resulting PDGs strictly preserve the same relation, namely $[\![\mathtt{P}]\!]_{PDG} < [\![\mathtt{P}]\!]_{PDG}^{\mathcal{U}}$. □

Theorem 4.10. $[\![\mathtt{P}]\!]_{PDG} = [\![\mathtt{P}]\!]_{PDG}^{\mathcal{U}}$ if and only if $\mathtt{P}$ does not contain fake dependencies.

Proof. The ($\Leftarrow$) direction comes from Lemma 4.7, while the ($\Rightarrow$) direction comes directly from Proposition 4.9. □

Note that, the presence of fake dependencies generates incompleteness since $[\![\mathtt{P}]\!]_{PDG}$ considers flow edges not concerning all the *used* variables. For instance in $y := x + z - x$ we have an edge from the definition of z to the expression $x + z - x$ but not from the definition of x, which is a fake dependence, and which is instead considered when abstracting by $\alpha_s^{\mathtt{Use}}$.

In the following, we show how we can use the incompleteness compressor on the above equation in order to characterise the potency of slicing obfuscation.

Theorem 4.11. Given a program $\mathtt{P}$, the incomplete compressor with respect to the domain $PDG_{\mathtt{P}}$ is the abstraction $\alpha_s^{\mathtt{F}}$ with $\mathtt{F}(l) = \mathit{Nodes}_{\mathtt{P}}$, adding all the possible flow edges between all the nodes of the PDG, namely between all the instructions of $\mathtt{P}$.

Proof. In order to prove the thesis, we have to identify precisely the portion in $\alpha_s^{\mathtt{F}}$ that approximated the flows. Let $s = \langle \sigma, \langle l, l' \rangle, \mathcal{P}_l, D_l \rangle$ be a state, $p_{\mathtt{P}}^{\mathsf{S}}$ denotes the function $\mathtt{p}$ specialised on the portion of the input consisting in σ, $\langle l, l' \rangle$, D_l, $\mathit{Nodes}(\mathcal{P}_l)$ and $\mathit{Control}(\mathcal{P}_l)$, namely the only unknown input is $\mathit{Flow}(\mathcal{P}_l)$. Then we observe that:

$$[\![\mathtt{P}]\!]_{PDG}(s) \neq [\![\mathtt{P}]\!]_{PDG}^{\alpha_s^{\mathtt{F}}}(s) \qquad \text{iff} \qquad p_{\mathtt{P}}^{\mathsf{S}}(\mathit{Flow}(\mathcal{P}_l))_{|\mathit{Flow}} \neq p_{\mathtt{P}}^{\mathsf{S}} \circ \alpha_f^{\mathtt{F}}(\mathit{Flow}(\mathcal{P}_l))_{|\mathit{Flow}}$$

Consider $\mathcal{UR}^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}},id}(id)$ on $uco(\mathit{Flow}_{\mathtt{P}})$, by Proposition 3.14 we have that

$$\mathcal{UR}^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}},id}(id) = \mathcal{M}(\mathit{Mirr}(id \sqcap R^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}}}(id)) \smallsetminus R^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}}}(id)) = \mathcal{M}(\mathit{Mirr}(\mathit{Flow}_{\mathtt{P}}) \smallsetminus R^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}}}(id))$$

where $R^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}}}(id) = \mathcal{M}(\bigvee_{\mathcal{P} \in PDG} \max({p_{\mathtt{P}}^{\mathsf{S}}}^{-1}(\downarrow \mathcal{P}))$ and $\bigvee$ is a shorthand for $\bigvee_{PDG}$. At this point, in order to understand these elements we observe that ${p_{\mathtt{P}}^{\mathsf{S}}}^{-1}$ goes back one step of execution for each PDG in $\downarrow\mathcal{P}$. Let us observe that, in order to compute the PDG semantics, we have to compute the fix point of $\mathtt{p}$, and therefore of $p_{\mathtt{P}}^{\mathsf{S}}$. This means that any PDG can be obtained as inverse image of a PDG, in particular, if a portion of the program $\mathtt{P}$ is missing then it is the image of the PDG where one more statement is executed, otherwise it is the fix point and therefore it is the image of itself. Therefore, also all the elements in $\mathit{Mirr}(\mathit{Flow}_{\mathtt{P}})$ can be inverse image of $p_{\mathtt{P}}^{\mathsf{S}}$, i.e.,

$$\mathit{Mirr}(\mathit{Flow}_{\mathtt{P}}) \smallsetminus R^{\mathscr{B}}_{\mathtt{P}}(id) = \varnothing \;\Rightarrow\; \mathcal{UR}^{\mathscr{B}}_{p_{\mathtt{P}}^{\mathsf{S}},id}(id) = \mathcal{M}(\varnothing) = \lambda X \subseteq \mathit{Flow}_{\mathtt{P}}.\, \mathit{Flow}_{\mathtt{P}}$$

This function corresponds to $\alpha_f^{\mathtt{F}}$, where $\mathtt{F}$ is defined as $\forall l \in \mathit{Nodes}_{\mathtt{P}}.\, \mathtt{F}(l) = \mathit{Nodes}_{\mathtt{P}}$, i.e., it is the function adding any kind of noise to the set of flow edges. We can conclude that the maximal possible noise about flow edges is characterised by this $\mathtt{F}$, and the corresponding state abstraction is $\alpha_s^{\mathtt{F}}$. □

The meaning of this theorem and also of its proof is that, in this case, the only possibility for adding noise is to enlarge the set of flows in the PDG computation of a program. Thus moving from the concrete set of dependency-based flow edges, to the set of all the possible flow edges.

Corollary 4.12. Let $\mathfrak{O}$ and obfuscation technique adding fake dependencies to a PDG, and let $\mathcal{S}$ be the function mapping programs to PDG. Then

$$\mathcal{P}\mathrm{ot}_{\mathfrak{O},\mathcal{S}} = \left\{\, \mathcal{A} \;\middle|\; \mathcal{A} \sqsubseteq \alpha_f^{\mathtt{F}},\ \text{where } \forall l \in \mathit{Nodes}_{\mathtt{P}}.\, \mathtt{F}(l) = \mathit{Nodes}_{\mathtt{P}} \,\right\}$$

It is worth noting that, once we have an approximated PDG $\mathcal{P}_{\mathtt{P}}^{\mathtt{F}}$ of a program $\mathtt{P}$, namely the PDG of $\mathtt{P}$ computed by considering the abstraction of flow edges $\mathtt{F}$ approximating $\mathtt{Use}$, a way to implement the added noise consists in transforming the program in a way such that: for each $x \in \mathtt{F}(l')$ with x defined in l ($\langle l, l' \rangle \in \mathit{Flow}(\mathcal{P}_{\mathtt{P}}^{\mathtt{F}})$) we transform $\mathtt{Stm}_{\mathtt{P}}(l')$ by adding fake dependencies which use the variable x. For instance by transforming an expression $\mathtt{e}$ in $\mathtt{e} + x - x$, or adding an opaque predicate on x. In this way, we generate a new program $\mathfrak{O}(\mathtt{P})$ such that $[\![\mathtt{P}]\!] = [\![\mathfrak{O}(\mathtt{P})]\!]$, which implies $\mathcal{P}_{\mathfrak{O}(\mathtt{P})} = \mathcal{P}_{\mathtt{P}}$, and such that its abstract PDG $\mathcal{P}^{\mathtt{Use}}_{\mathfrak{O}(\mathtt{P})}$ is precisely the final abstract PDG $\mathcal{P}_{\mathtt{P}}^{\mathtt{F}}$, namely $\mathcal{P}^{\mathtt{Use}}_{\mathfrak{O}(\mathtt{P})} = \mathcal{P}_{\mathtt{P}}^{\mathtt{F}} >_{PDG} \mathcal{P}_{\mathtt{P}}$. To conclude, the space of functions $\mathtt{F}$ provides the potency of slicing obfuscation since function $\mathtt{F}$ allows us to tune the amount of added noise.

4.4. Obfuscating Static Disassembly

In this section, we consider static disassembly and a typical code obfuscation technique used to induce a loss of precision in the disassembly process. By using the proposed framework we are able to provide both a

LINEAR SWEEP

```
global startAddr, endAddr;
proc DisasmLinear(addr)
  begin
  while (startAddr ≤ addr ≤ endAddr)
  do
  I := decode instruction at address addr;
  addr += length I;
  od
end

proc main()
  begin
  ep := program entry point;
  size := text section size;
  startAddr := ep;
  endAddr := ep + size;
  DisasmLinear(ep);
end
```

RECURSIVE TRAVERSAL

```
global startAddr, endAddr;
proc DisasmRec(addr)
  begin
  while (startAddr ≤ addr ≤ endAddr)
  do
  if (addr has been visited already) return;
  I:= decode instruction at address addr;
  mark addr as visited;
  if (I is a branch or function call)
    for each possible target t of I do
    DisasmRec(t);
    od
    return;
  else addr += length(I);
  od
  end
```

Fig. 8. Static disassembly

characterisation of the class of static disassembly algorithms for which the considered obfuscation is potent, and formal evidence of how the obfuscation should work to thwart static disassembly.

Disassembly refers to the process of recovering assembly code instructions from a machine code file. Indeed, the process of reverse engineering an executable program typically begins with disassembly, which translates machine code to assembly code. Hence, obfuscation techniques that aim at thwarting disassembly can be used to obstruct reverse engineering [LD03]. An executable file typically consists of a number of different sections and of a header describing these sections. These different sections, such as the text section, the read-only data section, etc., contain various sorts of information about the program. In particular, the header contains information about the program entry point and the total size or extent of its instructions. We consider here *static disassembly* algorithms that proceed by examining the file to disassemble without executing it. Given an executable file static disassembly algorithms begin by extracting the set of locations that are supposed to contain encoding of instructions, and then they proceed by decoding the hexadecimal values stored at these locations, thus recovering the corresponding assembly instructions. Indeed, the precision of static disassembly algorithms can be measured in terms of their precision in identifying the locations in the text section.

Two typical static disassembly algorithms are LINEAR SWEEP and RECURSIVE TRAVERSAL and they are reported in Figure 8. The LINEAR SWEEP algorithm proceeds by decoding all the hexadecimal values that can be found in the text section of the executable file, thus assuming that all the values in the text section are encoding of instructions. Indeed, it is well known that the main weakness of this algorithm is that it is prone to disassembly errors resulting from the misinterpretation of data that is embedded in the text section. The RECURSIVE TRAVERSAL algorithm is in general more precise since it identifies the locations to be disassembled by statically following the control flow of the executable. In particular, the RECURSIVE TRAVERSAL algorithm starts by decoding the hexadecimal value at the entry point of the text section and then it recursively decodes the hexadecimal value stored in locations that are possible successors of the decoded instruction. For these reasons, in order to obstruct static disassembly, researchers have developed obfuscation techniques that insert junk in the text section, where static disassembly algorithms, such as LINEAR SWEEP and RECURSIVE TRAVERSAL, assume to find instructions [LD03].

In the following, we show that the precision of a static disassembly algorithm can be expressed as a completeness problem with respect to the way that the disassembly algorithm approximates the set of locations that contain the encoding of program instructions. Moreover, we formally prove that in order to lose precision of the disassembly, namely completeness, we need to insert junk in the locations of the text section that do not store the encoding of instructions. We conclude by proving that the algorithm of LINEAR SWEEP is the $\mathscr{B}$-incomplete compressor of static disassembly. This means that LINEAR SWEEP represents the maximal imprecision that we can have in static disassembly when we insert junk in the text section.

In order to formally prove this we need to introduce some notation: $Loc \subseteq \mathbb{N}$ denotes the set of memory locations, Val denotes the set of possible hexadecimal values stored in a memory location, $mem : Loc \rightarrow Val$

denotes the memory map that specifies the hexadecimal value contained in a given location, $\mathbb{I}$ denotes the set of possible assembly instructions.

We assume that the decoding function keeps track of the locations where instructions and corresponding hexadecimal values are stored. Indeed, we define the decoding function as $decode : (Loc \to Val) \to (Loc \times \mathbb{I})$. Observe that if $mem(x) = \bot$ then $decode(x, \bot) = \emptyset$, meaning that if the memory location x does not contain an hexadecimal value then its decoding produces no assembly instructions. Given an executable file F we denote with $\mathtt{Text}_F \subseteq Loc$ the set of locations of the text section of F and with $\mathtt{Instr}_F \subseteq \mathtt{Text}_F$ the set of all and only the locations of the text section that contain the hexadecimal encoding of instructions of F. Thus, the precise disassembly of an executable F is defined as:

$$Disassembly(F) \overset{\text{def}}{=} \mathtt{StaticDis}_F(\mathtt{Instr}_F)$$

where function $\mathtt{StaticDis}_F : \wp(\mathtt{Text}_F) \to \wp(Loc \times \mathbb{I})$ is defined as follows:

$$\mathtt{StaticDis}_F(X) = \{\ decode(x, mem(x)) \mid\ x \in X\ \}$$

It is clear that, the main challenge of a static disassembly algorithm is to extract from an executable file F the set $\mathtt{Instr}_F$ of locations where instructions are really encoded. In general, a static disassembly algorithm will extract a sound approximation of $\mathtt{Instr}_F$, namely a superset of $\mathtt{Instr}_F$. This means that we can associate with each static disassembly algorithm a closure operator that models how the considered disassembly approximates the set of locations that are assumed to contain the encoding of instructions. Let $\eta \in uco(\wp(\mathtt{Text}_F))$ be the approximation of $\mathtt{Instr}_F$ associated to a static disassembly algorithm. Then, this algorithm computes a precise disassembly of an executable file F when $\mathtt{StaticDis}_F(\mathtt{Instr}_F) = \mathtt{StaticDis}_F(\eta(\mathtt{Instr}_F))$, namely when $\langle id, \eta\rangle$ is $\mathscr{B}$-complete for the computation of function $\mathtt{StaticDis}_F$. Observe that we have $\mathscr{B}$-completeness when η abstracts the set $\mathtt{Instr}_F$ by adding locations $x \in \mathtt{Text}_F \smallsetminus \mathtt{Instr}_F$ such that $mem(x) = \bot$, since in this case $decode(x, \bot) = \emptyset$ and this does not affect the result of disassembly. Of course we have precision when $\eta = id$.

Proposition 4.13. Given a static disassembly algorithm $\mathtt{A}$ and an executable file F, let $\eta_{\mathtt{A}} \in uco(\wp(\mathtt{Text}_F))$ be the closure operator that models how disassembly $\mathtt{A}$ approximates the locations that should contain the encoding of instructions of F, and let $id \in uco(\wp(Loc \times \mathbb{I}))$. Then, algorithm $\mathtt{A}$ computes a precise disassembly of F if and only if the domains $\langle id, \eta_{\mathtt{A}}\rangle$ are $\mathscr{B}$-complete for the computation of function $\mathtt{StaticDis}_F$.

Proof. Trivial by definition. □

The following result shows that the loss of precision in the static disassembly algorithms is due to junk inserted in locations of the text section, where the algorithms are supposed to find the encoding of instructions.

Proposition 4.14. Given a static disassembly algorithm $\mathtt{A}$ and an executable file F, let $\eta_{\mathtt{A}} \in uco(\wp(\mathtt{Text}_F))$ be the closure operator that models how disassembly $\mathtt{A}$ approximates the locations that should contain the encoding of instructions of F, and let $id \in uco(\wp(Loc \times \mathbb{I}))$. Then, algorithm $\mathtt{A}$ computes a precise disassembly of F if and only if $\forall x \in \eta_{\mathtt{A}}(\mathtt{Instr}_F) \smallsetminus \mathtt{Instr}_F : mem(x) = \bot$.

Proof. From Proposition 4.13 we have that algorithm $\mathtt{A}$ computes a precise disassembly of F if and only if $\mathtt{StaticDis}_F(\mathtt{Instr}_F) = \mathtt{StaticDis}_F(\eta_{\mathtt{A}}(\mathtt{Instr}_F))$. Recall that $decode(x, \bot) = \emptyset$ and that in general $\mathtt{Instr}_F \subseteq \eta_{\mathtt{A}}(\mathtt{Instr}_F)$. Thus, $\mathtt{StaticDis}_F(\mathtt{Instr}_F) = \mathtt{StaticDis}_F(\eta_{\mathtt{A}}(\mathtt{Instr}_F))$ holds if and only if $\mathtt{Instr}_F = \{\ x \in \mathtt{Text}_F \mid\ mem(x) \neq \bot\ \}$, namely if $\forall x \in \eta_{\mathtt{A}}(\mathtt{Instr}_F) \smallsetminus \mathtt{Instr}_F : mem(x) = \bot$. □

This precisely means that, in order to induce imprecision in the result of a static disassembly algorithm, it is sufficient to add noise (junk) exactly to those memory locations that are supposed to contain an instruction encoding, but which do not really contain such encodings. These location are those in $\eta_{\mathtt{A}}(\mathtt{Instr}_F) \smallsetminus \mathtt{Instr}_F$, added by the approximation induced by the considered static disassembly algorithm. Indeed, $\mathtt{StaticDis}_F(\mathtt{Instr}_F) \neq \mathtt{StaticDis}_F(\eta_{\mathtt{A}}(\mathtt{Instr}_F))$ when $\exists x \in \eta_{\mathtt{A}}(\mathtt{Instr}_F) \smallsetminus \mathtt{Instr}_F$ such that $mem(x) \neq \bot$, namely $decode(x, mem(x)) \neq \emptyset$.

Observe that, the closure operator associated with LINEAR SWEEP algorithm is $\eta_{LS} \in uco(\wp(\mathtt{Text}_F))$ such that $\eta_{LS}(X) = \mathtt{Text}_F$ for every $X \in \wp(\mathtt{Text}_F)$. This precisely models the fact that the LINEAR SWEEP algorithm approximates $\mathtt{Instr}_F$ with the entire text section. It is possible to prove that the $\mathscr{B}$-incomplete compressor of $\mathtt{StaticDis}_F$ with respect to (id, id) is precisely η_{LS}, namely the algorithm of LINEAR SWEEP. This means that, as expected, the LINEAR SWEEP algorithm induces the coarsest approximation of $\mathtt{Instr}_F$ thus inducing the maximal loss of precision in presence of junk added in the text section.

Proposition 4.15. $\mathcal{UR}^{\mathscr{B}}_{\mathtt{StaticDis}_F, id}(id) = \eta_{LS} \in \wp(\mathtt{Text}_F)$.

Proof. Observe that function $\mathtt{StaticDis}_F$ has inverse $\mathtt{StaticDis}_F^{-1} : \wp(Loc \times \mathbb{I}) \to \wp(\mathtt{Text}_F)$ defined as: $\mathtt{StaticDis}_F^{-1}(Y) = \{\ x \mid (x, I) \in Y\ \}$. We have to show that $\mathcal{UR}^{\mathscr{B}}_{\mathtt{StaticDis}_F, id}(id) = \eta_{LS}$. By definition we have that:

$$\mathcal{UR}^{\mathscr{B}}_{\mathtt{StaticDis}_F, id}(id) = \mathcal{M}(Mirr(id \sqcap \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y))) \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y)))$$

It is clear that id is the most concrete closure and therefore:

$$\mathcal{M}(Mirr(id \sqcap \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y))) \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y))) =$$
$$\mathcal{M}(Mirr(id) \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y)))$$

by definition of meet-irreducible elements we have that:

$$\mathcal{M}(Mirr(id) \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y))) = \mathcal{M}(\{\ \mathtt{Text}_F \smallsetminus \{x\} \mid x \in \mathtt{Text}_F\ \} \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y)))$$

Function $\mathtt{StaticDis}_F^{-1}$ is monotone and therefore:

$$\mathcal{M}(\{\ \mathtt{Text}_F \smallsetminus \{x\} \mid x \in \mathtt{Text}_F\ \} \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(\downarrow y))) =$$
$$\mathcal{M}(\{\ \mathtt{Text}_F \smallsetminus \{x\} \mid x \in \mathtt{Text}_F\ \} \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(y)))$$

Since we consider every possible element of $\wp(Loc \times \mathbb{I})$ we have that for every meet-irreducible element X of $\wp(\mathtt{Text}_F)$ there exist an element $y \in \wp(Loc \times \mathbb{I})$ such that $\mathtt{StaticDis}_F^{-1}(y) = X$, thus:

$$\mathcal{M}(\{\ \mathtt{Text}_F \smallsetminus \{x\} \mid x \in \mathtt{Text}_F\ \} \smallsetminus \mathcal{M}(\bigcup_{y \in id} \mathtt{StaticDis}_F^{-1}(y))) = \mathcal{M}(\emptyset)$$

By definition of Moore closure we have that $\mathcal{M}(\emptyset) = \lambda X.\mathtt{Text}_F$ and by definition this closure is precisely η_{LS}, and this concludes the proof. □

Corollary 4.16. Let $\mathfrak{O}$ be an obfuscating technique that adds junk in the locations of the text section that do not encode instructions, let $\mathcal{S}$ be the $\mathtt{StaticDis}_F$ function then:

$$\mathcal{P}ot_{\mathfrak{O},\mathcal{S}} = \{\ \mathcal{A} \mid id \sqsubset \mathcal{A} \sqsubseteq \eta_{LS}\ \}$$

Indeed, the above result proves that all the static disassembly algorithms whose characterising closure $\eta_{\mathtt{A}} \in uco(\wp(\mathtt{Text}_F))$ is such that $id \sqsubset \eta_{\mathtt{A}} \sqsubseteq \eta_{LS}$ produces an imprecise disassembly when junk is inserted in the locations of the text section that do not encode instructions. In other words the obfuscation that inserts junk in the text section is potent with respect to all the static disassembly algorithms that induce an approxiamtion of $\mathtt{Instr}_F$ between $id(\mathtt{Instr}_F)$ excluded and $\eta_{LS}(\mathtt{Instr}_F)$ included. Observe, for example, that the closure operator $\eta_{RT} \in uco(\wp(\mathtt{Text}_F))$ associated to the RECURSIVE TRAVERSAL static disassembly algorithm is such that $id \sqsubseteq \eta_{RT} \sqsubseteq \eta_{LS}$, meaning that the obfuscation that inserts junk in the text section is potent also with respect to the RECURSIVE TRAVERSAL algorithm, which is however more precise than LINEAR SWEEP.

5. Discussion: Towards incompleteness driven obfuscation design

We introduced the notion of incomplete compressor for an abstract domain and proved that, under non-restrictive hypotheses, it induces maximal incomplete abstract interpretations. The incomplete compression provides an adequate model for formally characterising the potency of an obfuscating program transformation. In particular, the incomplete compression removes precisely those elements of the abstract domain which are necessary to achieve a precise (complete) abstract interpretation of the program. These elements drive the construction of the obfuscating transformation. The idea is that the obfuscation should make the original abstract interpretation of the transformed program as imprecise as if we perform the abstract interpretation of the source code on the compressed abstract domain. This is what we observed in the obfuscation of program slicing and static disassembly, as shown in Section 4, where the domain compression corresponds respectively to augment dependencies and add junk instructions in specific locations. These features can be injected by simple program transformations which obfuscate the code making program slicing and code disassembly ineffective.

This idea can be partially automated in combination with the definition of obfuscating transformers as specialised interpreters, as introduced in [JGM12]. The obfuscating transformation is here the result of the specialisation of an interpreter modified in order to induce incompleteness. Correctness is ensured here by the first Futamura projection [JGS93], while obfuscation is given by distorting the interpreter. An interpreter is a *self-interpreter* if it written precisely in the interpreted language. This choice is connected with the process of specialization. Indeed, specialization (also known as partial evaluation) means to partially evaluate a program on some known inputs, namely the specialised program [JGS93] is precisely the same but with some instantiated computations. Hence, an interpreter specialised on a program is precisely the interpreter where all the computations concerning the input program are instantiated. Suppose that the program to obfuscate is written in the language $\mathcal{L}$, then if we want the obfuscated program (i.e., the specialised interpreter) to be written in the same language $\mathcal{L}$, then the interpreter `interp` has to be written precisely in $\mathcal{L}$, namely it has to be a self-interpreter[7]. The obfuscation of `P` can be obtained by modifying the self-interpreter `interp` in order to force abstract interpretation to deal with operations that induce incompleteness in the attacker. This yields, by specialization, a modified program $\texttt{P}' := [\![\texttt{spec}]\!](\texttt{interp}^+, \texttt{P})$ which is precisely the obfuscation of `P`. The incomplete compressor suggests here a way to design $\texttt{interp}^+$: The information removed by the compressor is precisely the noise that $\texttt{interp}^+$ has to introduce in the interpretation of `P`. Although intuitively clear, a calculational design (in the style of [Cou99]) of the distorted interpreter $\texttt{interp}^+$ from the incomplete compressor of the attacker is still a major challenge in fully automating the design of obfuscators.

References

[BGI+12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[BJ72] T.S. Blyth and M.F. Janowitz. *Residuation theory.* Pergamon Press, 1972.

[BN15] F. Brunton and H. Nissenbaum. *Obfuscation – A User's Guide for Privacy and Protest.* MIT Press, 2015.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, 1977.

[CC79b] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, 1979.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comput.*, 2(4):511–547, 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation (Invited Paper). In M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Internat. Symp. on Programming Language Implementation and Logic Programming* (*PLILP '92*), volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.

[CFG+95] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. In A. Mycroft, editor, *Proceedings of the 2nd International Static Analysis Symposium* (*SAS '95*), volume 983 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1995.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CN09] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 2009.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors,

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

Calculational System Design, volume 173, pages 421–505. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, Amsterdam, 1999.

[CTL98] C. Collberg, C. D. Thomborson, and D. Low. Manufactoring cheap, resilient, and stealthy opaque constructs. In *Proc. of Conf. Record of the 25st ACM Symp. on Principles of Programming Languages* (*POPL '98*), pages 184–196. ACM Press, 1998.

[DG05] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proc. of the 32nd International Colloquium on Automata, Languages and Programming* (*ICALP '05*), volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336. Springer-Verlag, 2005.

[DG09] Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[DTM07] S. Drape, C. Thomborson, and A. Majumdar. Specifying imperative data obfuscations. In J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, editors, *ISC - Information Security*, volume 4779 of *Lecture Notes in Computer Science*, pages 299 – 314. Springer Verlag, 2007.

[FR96] G. Filé and F. Ranzato. Complementation of abstract domains made easy. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming* (*JICSLP '96*), pages 348–362. The MIT Press, 1996.

[GGH$^+$13] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[Gia08] R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*, pages 7–20. IEEE Press., 2008.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. on Software Engineering*, 17(8):751–761, 1991.

[GLR15] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 261–273. ACM, 2015.

[GM12] R. Giacobazzi and I. Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 129–145, 2012.

[GQ01] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th Internat. Static Analysis Symp. (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 2001.

[GR97] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. of the 24th Internat. Colloq. on Automata, Languages and Programming* (*ICALP '97*), volume 1256 of *Lecture Notes in Computer Science*, pages 771–781. Springer-Verlag, 1997.

[GR98] R. Giacobazzi and F. Ranzato. Uniform closures: order-theoretically reconstructing logic program semantics and abstract domain refinements. *Inform. and Comput.*, 145(2):153–190, 1998.

[GRS00] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.

[HRB90] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[Jan67] M. F. Janowitz. Residuated closure operators. *Portug. Math.*, 26(2):221–252, 1967.

[JGM12] N. D. Jones, R. Giacobazzi, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In O. Kiselyov and S. Thompson, editors, *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, pages 63 – 72. ACM Press, 2012.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., 1993.

[LD03] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[LL09] V. Laviron and F. Logozzo. Refining abstract interpretation-based static analyses with hints. In *Proc. of APLAS'09*, volume 5904 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, 2009.

[MDT07] A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 70–81. ACM, 2007.

[MG15] I. Mastroeni and R. Giacobazzi. Weakening residuation in adjoining closures. *Order*, To appear., 2015. RR 95/2015, `http://hdl.handle.net/11562/925745`.

[MZ08] I. Mastroeni and D. Zanardini. Data dependencies and program slicing: From syntax to abstract semantics. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'08)*, pages 125 – 134. ACM Press, 2008.

[Rep91] T. Reps. Algebraic properties of program integration. *Sci. Comput. Program.*, 17:139–215, 1991.

[RT96] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Proceedings of the Dagstuhl seminar on Partial evaluation*, pages 409–429. Springer-Verlag, 1996.

[RY89] T. Reps and W. Yang. The semantics of program slicing and program integration. In J. Diaz and F. Orejas, editors, *Proc. of the Colloq. on Current Issues in Programming Languages*, volume 352 of *Lecture Notes in Computer Science*, pages 360–374. Springer-Verlag, 1989.

[Wei81] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.