Towards a unifying framework for tuning analysis precision by program transformation

Mila Dalla Preda
Dipartimento di Informatica, University of Verona, Italy
mila.dallapreda@univr.it

Abstract

Static and dynamic program analyses attempt to extract useful information on program's behaviours. Static analysis uses an abstract model of programs to reason on their runtime behaviour without actually running them, while dynamic analysis reasons on a test set of real program executions. For this reason, the precision of static analysis is limited by the presence of false positives (executions allowed by the abstract model that cannot happen at runtime), while the precision of dynamic analysis is limited by the presence of false negatives (real executions that are not in the test set). Researchers have developed many analysis techniques and tools in the attempt to increase the precision of program verification. Software protection is an interesting scenario where programs need to be protected from adversaries that use program analysis to understand their inner working and then exploit this knowledge to perform some illicit actions. Program analysis plays a dual role in program verification and software protection: in program verification we want the analysis to be as precise as possible, while in software protection we want to degrade the results of analysis as much as possible. Indeed, in software protection researchers usually recur to a special class of program transformations, called code obfuscation, to modify a program in order to make it more difficult to analyse while preserving its intended functionality. In this setting, it is interesting to study how program transformations that preserve the intended behaviour of programs can affect the precision of both static and dynamic analysis. While some works have been done in order to formalise the efficiency of code obfuscation in degrading static analysis and in the possibility of transforming programs in order to avoid or increase false positives, less attention has been posed to formalise the relation between program transformations and false negatives in dynamic analysis. In this work we are setting the scene for a formal investigation of the syntactic and semantic program features that affect the presence of false negatives in dynamic analysis. We believe that this understanding would be useful for improving the precision of existing dynamic analysis tools and in the design of program transformations that complicate the dynamic analysis.

To Maurizio on its 60th birthday!

2012 ACM Subject Classification Security and privacy → Software and application security → Software reverse engineering

Keywords and phrases Program analysis, analysis precision, program transformation, software protection, code obfuscation

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.4

Funding The research has been partially supported by the project "Dipartimenti di Eccellenza 2018-2020" funded by the Italian Ministry of Education, Universities and Research (MIUR).

1 Introduction

Program analysis refers, in general, to any examination of programs that attempts to extract useful information on program's behaviours (semantics). As known from the Rice theorem, all nontrivial extensional properties of program's semantics are undecidable in the general case. This means that any automated reasoning on software has to involve some kind of approximation. Programs can be analysed either statically or dynamically. Static program

Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 4; pp. 4:1–4:22
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

analysis reasons about the behaviour of programs without actually running them. Typically, static analysis builds an abstract model that over-approximates the possible program's behaviours to examine program properties. This guarantees soundness: what can be derived from the analysis of the abstract model holds also on the concrete execution of the program. The converse does not hold in general due to the presence of *false positives*: spurious behaviours allowed by the abstract model that do not correspond to any real program execution. Static analysis has proved its usefulness in many fields of computer science like in optimising compilers for producing efficient code, for automatic error detection and for the automatic verification of desired program properties (e.g., functional properties and security properties) [21]. Many different static analysis approaches exists, as for example model checking [7], deductive verification [33] and abstract interpretation [12]. In particular, abstract interpretation provides a formal framework for reasoning on behavioural program properties where many static analysis techniques can be formalised. In the rest of this paper we focus on those static analysis that can be formalised in the abstract interpretation framework. Dynamic program analyses, such as program testing [1], runtime monitoring and verification [4], consider an under-approximation of program behaviour as they focus their analysis on a specific subset of possible program executions. In this paper when we speak of dynamic analysis we mainly refer to program testing. Testing techniques start by concretely executing programs on an input set and the so obtained test set of concrete executions is inspected in order to reason on program's behaviour (e.g., reveal failures or vulnerabilities). It is well known that dynamic analysis can precisely detect the presence of failures but cannot guarantee their absence, due to the presence of *false negatives*: concrete program behaviours that do not belong to the test set. There is a famous quote by Dijksta that states that "Program testing can be used to show the presence of bugs, but never to show their absence!". Since it is not possible to guarantee the absence of failures we have to accept the fact that whenever we use software we incur in some risk. Software testing is widely used to reveal possible software failures, to reduce the risk related to the use of software and to increase the quality of software by deciding if the behaviour of software is acceptable in terms of reliability, safety, maintainability, security, and efficiency [1].

Static analysis computes an over-approximation of program semantics, while dynamic analysis under-approximates program semantics. In both cases, we have a decidable evaluation of the semantic property of interest on an approximation of program semantics. For this reason what we can automatically conclude regarding the behavioural properties of programs has to take into account false positives for static analysis and false negatives for dynamic analysis. Static analysis is precise when it is *complete* (no false positives) and this relates to the well studied notion of completeness in abstract interpretation [12, 14, 23]. The intuition is that static analysis is complete when the details lost by the abstract model are not relevant for reasoning on the semantic property of interest. Dynamic analysis is precise when it is *sound* (no false negatives) and this happens when the executions in the test set exhibit all the behaviours of the program that are relevant with respect to the semantic property of interest. This means that the under-approximation of program semantics considered by the dynamic analysis allows us to precisely observe the behavioural property of interest. The essential problem with dynamic analysis is that it is impossible to test with all inputs since the input space is generally infinite. In this context, *coverage criteria* provide structured, practical ways to search the input space and to decide which input set to use. The rationale behind coverage criteria is to partition the input space in order to maximise the executions present in the tests set that are relevant for the analysis of the semantic property of interest. Coverage criteria are useful in supporting the automatic generation of input sets and in providing useful rules for deciding when to terminate the generation of the test set [1].

Program analysis has been originally developed for verifying the correctness of programs and researchers have put a great deal of effort in developing efficient and precise analysis techniques and tools that try to reduce false positives and false negatives as much as possible. Indeed, analysis precision relates to the ability of identifying failures and vulnerabilities that may lead to unexpected behaviours, or that may be exploited by an adversary for malicious purposes. For this reason the main goal of researchers has been to improve the precision and efficiency of both static and dynamic analysis tools.

Software protection is another interesting scenario where program analysis plays a central role but in a dual way. Today, software and the assets embedded in it are constantly under attack. This is particularly critical for those software applications that run in an untrusted environment in a scenario known as MATE (Man-At-The-End) attacks. In this setting, attackers have full control over, and white-box access to, the software and the systems on which the software is running. Attackers can use a wide range of analysis tools such as disassemblers, code browsers, debuggers, emulators, instrumentation tools, fuzzers, symbolic execution engines, customised OS features, pattern matchers, etc. to inspect, analyse and alter software and its assets. In such scenarios, software protection becomes increasingly important to protect the assets, even against MATE attacks. For industry, in many cases the deployment of software-based defense techniques is crucial for the survival of their businesses and eco-systems. In the software protection scenario, program analysis can be used by adversaries to reverse engineer proprietary code and then illicitly reuse portions of the code or tamper with the code in some unauthorised way. Here, in order to protect the intellectual property and integrity of programs we have to force the analysis to be imprecise or so expensive to make it impractical for the adversary to mount an attack.

To address this problem, researchers have developed software-based defense techniques, called *code obfuscations*, that transform programs with the explicit intent of complicating and degrading program analysis [9]. The idea of code obfuscation techniques is to transform a program into a functionally equivalent one that is more difficult (ideally impossible) for an analyst to understand. As well as for program analysis also for code obfuscation we have an important negative result from Barak et al. [3] that proves the impossibility of code obfuscation. Note that, this result states the impossibility of an ideal obfuscator that obfuscates every program by revealing only the properties that can be derived from its I/O semantics. Besides the negative result of Barak et al., in recent decades, we have seen a big effort in developing and implementing new and efficient obfuscation strategies [8]. Of course, these obfuscating techniques introduce a kind of practical obfuscators weakening the ideal obfuscator of Barak et al. in different ways, and which can be effectively used in real application protection in the market. For example, these obfuscators may work only for a certain class of programs, or may be able to hide only certain properties of programs (e.g., control flow). Indeed, the attention on code obfuscation poses the need to deeply understand what we can obfuscate, namely which kind of program properties we can hide by inducing imprecision in their automatic analysis.

A recent survey on the existing code obfuscation techniques shows the efficiency of code obfuscation in degrading the results of static analysis, while existing code obfuscation techniques turn out to be less effective against dynamic analysis [31]. Consider, for example, the well known control flow obfuscation based on the insertion of opaque predicates. An opaque predicate is a predicate whose constant value is known to the obfuscation, while it is difficult for the analyst to recognise such constant value [9]. Consider the program whose control flow graph is depicted on the left of Figure 1 where we have three blocks of sequential

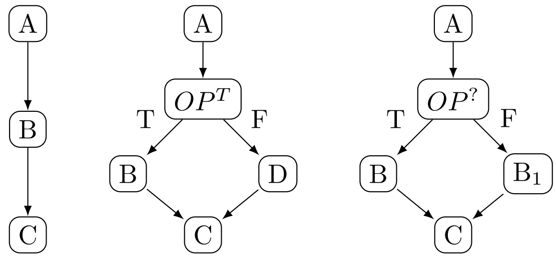

Figure 1 Code obfuscation

instructions A, B and C executed in the order specified by the arrows $A \to B \to C$. Let OP^T denote a true opaque predicate, namely a predicate that always evaluates to *true*. In the middle of Figure 1 we can see what happens to the control flow graph when we insert a true opaque predicate: block D has to be considered in the static analysis of the control flow even if it is never executed at runtime. Thus, $A \to OP^T \to D \to C$ is a false positive path added by the obfuscating transformation to the static analysis, while no imprecision is added to dynamic analysis since all real executions follow the path $A \to OP^T \to B \to C$. On the right of Figure 1 we have the control flow graph of the program obtained inserting an unknown opaque predicate. An unknown opaque predicate $OP^?$ is a predicate that sometimes evaluates to *true* and sometimes evaluates to *false*. These predicates are used to diversify program execution by inserting in the true and false branches sequences of instructions that are syntactically different but functionally equivalent (e.g. blocks B and B_1) [9]. Observe that this transformation adds confusion to dynamic analysis: a dynamic analyser has now to consider more execution traces in order to cover all the paths of the control flow graph. Indeed, if the dynamic analysis observes only traces that follow the original path $A \to OP^? \to B \to C$ it may not be sound as it misses the traces that follow $A \to OP^? \to B_1 \to C$ (false negative).

The abstract interpretation framework has been used to formalise, prove and compare the efficiency of code obfuscation techniques in confusing static analysis [17, 25] and to derive strategies for the design of obfuscating techniques that hamper a specific analysis [19]. The general idea is that code obfuscation confuses static analysis by exploiting its conservative nature, and by modifying programs in order to increase its imprecision (adding false positives) while preserving the program intended behaviour. Observe that, in general, the imprecision added by these obfuscating transformations to confuse a static analyser is not able to confuse a dynamic attacker that cannot be deceived by false positives. This is the reason why common deobfuscation approaches often recur to dynamic analysis to reverse engineer obfuscated code [5, 10, 32, 34].

It is clear that to complicate dynamic analysis we need to develop obfuscation techniques that exploit the Achilles heel of dynamic analysis, namely false negatives. In the literature, there are some defense techniques that focus on hampering dynamic analysis [2, 27, 28, 30]. What is still missing is a general framework where it is possible to formalise, prove and discuss the efficiency of these transformations in complicating dynamic analysis in terms of the imprecision (false negatives) that they introduce. As discussed above the main challenge for dynamic analysis is the identification of a suitable input set for testing program's behaviour. In order to automatically build a suitable input set, the analysts either design an input generation tool or an input recogniser tool. In both cases, they need a coverage criterion that defines the inputs to be considered and when to terminate the definition of the input set. Ideally, the coverage criterion is chosen in order to guarantee that the test set precisely reveals the semantic property under analysis (no false negatives). However, to the best of our knowledge, there is no formal guarantee that a coverage criterion ensures the absence of false negatives with respect to a certain analysis. If hampering static analysis means to increase the presence of false positives, hampering dynamic analysis means to complicate the automatic construction of a suitable input set for a given coverage criterion. In order to formally reason on the effects that code obfuscation has on the precision of dynamic analysis it is important to develop a general framework, analogous to the one based on program semantics and abstract interpretation that formalises the relation between dynamic analysis and code obfuscation. Thus, we need to develop a framework where we can (1) formally specify the relation between the coverage criterion used and the semantic property that we are testing, (2) define when a program transformation complicates the construction of an input set that has to satisfy a given coverage criterion, (3) derive guidelines for the design of obfuscating transformations that hamper the dynamic analysis of a given program property. This formal investigation will allow us to better understand the potential and limits of code obfuscation against dynamic program analysis.

In the following we provide a unifying view of static and dynamic program analysis and of the approaches that researchers use to tune the precision of these analysis. From this unifying overview it turns out that while the relation between the precision of static program analysis and program transformations has been widely studied, both in the software verification and in the software protection scenario, less attention has been posed to the formal investigation of the effects that code transformations have on the precision of program testing. We start to face this problem by showing how it is possible to formally compare and relate coverage criterion, semantic property under testing and false negatives for a specific class of program properties. This discussion leads us to the identification of important and interesting new research directions that would lead to the development of the above mentioned formal framework for reasoning about the effects of program transformations on the precision of dynamic analysis. We believe that this formal reasoning would find interesting applications both in the software verification and in the software protection scenario.

Structure of the paper: In Section 2 we provide some basic notions. In Section 3 we discuss possible techniques for improving the precision of the analysis: Section 3.1 revise the existing and ongoing work in transforming properties and programs toward completeness of static analysis, while Section 3.2 provides the basis for a formal framework for reasoning on possible property and program transformations to achieve soundness in dynamic analysis, these are preliminary results some of which have been recently published in [18]. Section 4 shows how the techniques used to improve analysis precision could be used in the software protection scenario to prove the efficiency of software protection techniques. The use of this formal reasoning for proving the efficiency of software protection techniques against static analysis is known, while it is novel for dynamic analysis. The paper ends with a discussion on the open research challenges that follow from this work.

2 Preliminaries

Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \times T$ the Cartesian product of S and T, with $S \subset T$ strict inclusion, with $S \subseteq T$ inclusion, with $S \subseteq_F T$ the fact that S is a finite subset of T. $\langle C, \leq_C, \vee_C, \wedge_C, \top_C, \bot_C \rangle$ denotes a complete lattice on the set C, with ordering $\leq_C$, least upper bound (*lub*) $\vee_C$, greatest lower bound (*glb*) $\wedge_C$, greatest element (top) $\top_C$, and least element (bottom) $\bot_C$ (the subscript $_C$ is omitted when the domain is clear from the context). Let C and D be complete lattices. Then, $C \xrightarrow{\text{m}} D$ and

$C \xrightarrow{c} D$ 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. Let $f : C \to C$ be a function on a complete lattice C, we denote with $\mathit{lfp}(f)$ the least fix-point, when it exists, of function f on C. The well-known Knaster-Tarski's theorem states that any monotone operator $f : C \xrightarrow{m} C$ on a complete lattice C admits a least fix point. 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: $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$.

Program Semantics: Let us consider the set *Prog* of possible programs and the set Σ of possible program states. A program state $s \in \Sigma$ provides a snapshot of the program and memory content during the execution of the program. Given a program P we denote Init_P the set of its initial states. We use Σ^* to denote the set of all finite and infinite sequences or traces of states ranged over by σ. Given a trace $\sigma \in \Sigma^*$ we denote with $\sigma_0 \in \Sigma$ the first element of sequence σ and with σ_f the final state of σ if σ is finite. Let $\tau \subseteq \Sigma \times \Sigma$ denote the transition relation between program states, thus $(s, s') \in \tau$ means that state s' can be obtained from state s in one computational step. The *trace semantics* of a program P is defined, as usual, as the least fix-point computation of function $\mathcal{F}_P : \wp(\Sigma^*) \to \wp(\Sigma^*)$ [11]:

$$\mathcal{F}_P(X) \stackrel{\text{def}}{=} \mathit{Init}_P \cup \left\{ \sigma s_i s_{i+1} \;\middle|\; (s_i, s_{i+1}) \in \tau, \sigma s_i \in X \right\}$$

The trace semantics of P is $\llbracket P \rrbracket \stackrel{\text{def}}{=} \mathit{lfp}(\mathcal{F}_P) = \bigcup_{i\in\mathbb{N}} \mathcal{F}_P^i(\bot_C)$. $\mathit{Den}\llbracket P \rrbracket$ denotes the denotational (finite) semantics of program P which abstracts away the history of the computation by observing only the input-output relation of finite traces: $\mathit{Den}\llbracket P \rrbracket \stackrel{\text{def}}{=} \{\sigma \in \Sigma^+ \mid \exists \eta \in \llbracket P \rrbracket : \eta_0 = \sigma_0, \eta_f = \sigma_f\}$.

Concrete domains are collections of computational objects where the concrete semantics is computed, while abstract domains are collections of approximate objects, representing properties of concrete objects in a domain-like structure. It is possible to interpret the semantics of programs on abstract domains thus approximating the computation with respect to the property expressed by the abstract domain. The relation between concrete and abstract domains can be equivalently specified in terms of Galois connections (GC) or upper closure operators in the abstract interpretation framework [12, 13]. The two approaches are equivalent, modulo isomorphic representations of the domain object. A GC is a tuple (C, α, γ, A) where C is the concrete domain, A is the abstract domain and $\alpha : C \to A$ and $\gamma : A \to C$ are respectively the abstraction and concretisation maps that give rise to an adjunction: $\forall a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Abstract domains can be compared with respect to their relative degree of precision: if A_1 and A_2 are abstractions of a common concrete domain C, A_1 is more precise than A_2, denoted $A_1 \sqsubseteq A_2$ when $\forall a_2 \in A_2, \exists a_1 \in A_1 : \gamma_1(a_1) = \gamma_2(a_2)$, namely if $\gamma_2(A) \subseteq \gamma_1(A)$. An upper closure operator on a complete lattice C is an operator $\rho : C \to C$ that is monotone, idempotent, and extensive ($\forall x \in C : x \leq_C \rho(x)$). Closures are uniquely determined by their fix-points $\rho(C)$. If (C, α, γ, A) is a GC then $\rho = \gamma \circ \alpha$ is the closure associated to A, such that $\rho(C)$ is a complete lattice isomorphic to A. The closure $\gamma \circ \alpha$ associated to the abstract domain A can be thought of as the logical meaning of A in C, since this is shared by any other abstract representation for the objects of A. Thus, the closure operator approach is convenient when reasoning about properties of abstract domains independently from the representation of their objects. We denote with $\mathit{uco}(C)$ the set of upper closure operators over C. If C is a complete lattice then $\mathit{uco}(C)$ is a complete lattice where closure are ordered with respect to their relative precision $\rho_1 \sqsubseteq \rho_2 \Leftrightarrow \rho_2(C) \subseteq \rho_1(C)$ which corresponds to the ordering of abstract domains.

The abstract semantics of a program P on the abstract domain $\rho \in uco(\wp(\Sigma^*))$, denoted as $[\![P]\!]^\rho$, is defined as the fix-point computation of function $\mathcal{F}_P^\rho : \rho(\wp(\Sigma^*)) \to \rho(\wp(\Sigma^*))$ where $\mathcal{F}_P^\rho \stackrel{\text{def}}{=} \rho \circ \mathcal{F}_P \circ \rho$ is the best correct approximation of function $\mathcal{F}_P$ on the abstract domain $\rho(\wp(\Sigma^*))$, namely $[\![P]\!]^\rho \stackrel{\text{def}}{=} \mathit{lfp}(\mathcal{F}_P^\rho) = \bigcup_{i\in\mathbb{N}} \mathcal{F}_P^\rho(\bot_{\rho(C)})$. Given the equivalence between GC and closures, the abstract semantics can be equivalently specified in terms of abstract traces in the corresponding abstract domain and in the following we denote the abstract semantics either with $[\![P]\!]^\rho$ or with $[\![P]\!]^A$ where (C, α, γ, A) is a GC and $\rho = \gamma \circ \alpha$.

Equivalence Relations: Let $\mathcal{R}$ be a binary relation $\mathcal{R} \subseteq C \times C$ on a set C, given $x, y \in C$ we denote with $(x, y) \in \mathcal{R}$ the fact that x is in relation $\mathcal{R}$ with y. $\mathcal{R} \subseteq C \times C$, is an *equivalence relation* if $\mathcal{R}$ is reflexive $\forall x \in C : (x, x) \in \mathcal{R}$, symmetric $\forall x, y \in C : (x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$ and transitive $\forall x, y, z \in C : (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$. Given a set C equipped with an equivalence relation $\mathcal{R}$, we consider for each element $x \in C$ the subset $[x]_\mathcal{R}$ of C containing all the elements of C in equivalence relation with x, i.e., $[x]_\mathcal{R} = \{y \in C \mid (x, y) \in \mathcal{R}\}$. The sets $[x]_\mathcal{R}$ are called equivalence classes of C wrt relation $\mathcal{R}$ and they induce a partition of the set C, namely $\forall x, y \in C : [x]_\mathcal{R} = [y]_\mathcal{R} \vee [x]_\mathcal{R} \cap [y]_\mathcal{R} = \emptyset$ and $\cup\{[x]_\mathcal{R} \mid x \in C\} = C$. The partition of C induced by the relation $\mathcal{R}$ is denoted by $C/_\mathcal{R}$. Let $Eq(C)$ be the set of equivalence relations on the set C. The set of equivalence relations on C form a lattice $\langle Eq(C), \preceq, \sqcap_{Eq}, \sqcup_{Eq}, id, top\rangle$ where id is the relation that distinguishes all the elements in C, top is the relation that cannot distinguish any element in C, and: $\mathcal{R}_1 \preceq \mathcal{R}_2$ iff $\mathcal{R}_1 \subseteq \mathcal{R}_2$ iff $(x, y) \in \mathcal{R}_1 \Rightarrow (x, y) \in \mathcal{R}_2$, $\mathcal{R}_1 \sqcap_{Eq} \mathcal{R}_2 = \mathcal{R}_1 \cap \mathcal{R}_2$, namely $(x, y) \in \mathcal{R}_1 \sqcap_{Eq} \mathcal{R}_2$ iff $(x, y) \in \mathcal{R}_1 \wedge (x, y) \in \mathcal{R}_2$; $\mathcal{R}_1 \sqcup_{Eq} \mathcal{R}_2$ it is such that $(x, y) \in \mathcal{R}_1 \sqcup_{Eq} \mathcal{R}_2$ iff $(x, y) \in \mathcal{R}_1 \vee (x, y) \in \mathcal{R}_2$. When $\mathcal{R}_1 \preceq \mathcal{R}_2$ we say that $\mathcal{R}_1$ is a refinement of $\mathcal{R}_2$. Given a subset $S \subseteq C$, we denote with $\mathcal{R}_{|_S} \in Eq(S)$ the restriction of relation $\mathcal{R}$ to the domain S.

The relation between closure operators and equivalence relations has been studied in [29]. Each closure operator $\rho \in uco(\wp(C))$ induces an equivalence relation $\mathcal{R}^\rho \in Eq(C)$ where $(x, y) \in \mathcal{R}^\rho y$ iff $\rho(\{x\}) = \rho(\{y\})$ and viceversa, each equivalence relation $\mathcal{R} \in Eq(C)$ induces a closure operator $\rho^\mathcal{R} \in uco(\wp(C))$ where $\rho^\mathcal{R}(\{x\}) = [x]_\mathcal{R}$ and $\rho^\mathcal{R}(X) = \bigcup_{x\in X} [x]_\mathcal{R}$. Of course, there are many closures that induce the same partition on traces and these closures carry additional information other than the underlying state partition, and this additional information that allows us to distinguish them is lost when looking at the induced partition. Indeed, it holds that given $\mathcal{R} \in Eq(C)$ the corresponding closure is such that $\rho^\mathcal{R} = \sqcap\{\rho \mid \mathcal{R}^\rho = \mathcal{R}\}$. The closures in $uco(\wp(C))$ defined form a partition $\mathcal{R} \in Eq(C)$ are called *partitioning* and they identify a subset of $uco(\wp(C))$: $\{\rho^\mathcal{R} \in uco(\wp(C)) \mid \mathcal{R} \in Eq(C)\} \subseteq uco(\wp(C))$ [29].

3 On the precision of program analysis

As argued above program analysis has been originally developed for program verification, namely to ensure that programs will actually behave as expected. Besides the impossibility result of the Rice theorem, a multitude of analysis strategies have been proposed [21]. Indeed, by tuning the precision of the behavioural feature that we want to analyse it is possible to derive an analysable semantic property that, while loosing some details of program's behaviour, may still be of practical interest [12, 14]. We are interested in semantic program properties, namely in properties that deal with the behaviour of programs, but the possibility of precisely analysing such properties depends also on the way in which programs are written. This means that there are programs that are easier to analyse than others with respect to a certain property [6]. Thus, program transformations that preserve the program's intended functionality can affect the precision of the results of the same analysis on the original and

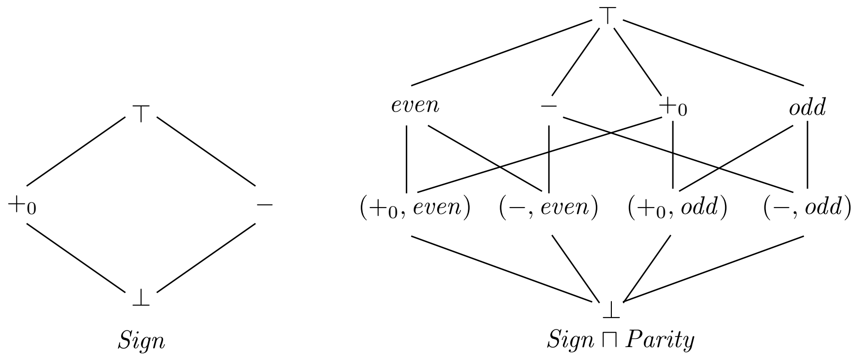

Figure 2 Abstract domain of Sign and Sign ⊓ Parity

transformed program.

3.1 Static Analysis

Precision in static program analysis means completeness, namely absence of false positives. This means that the noise introduced by the abstract model used for static program analysis does not introduce imprecision with respect to the property under analysis. Consider for example program P on the left of Figure 3 that, given an integer value a, returns its absolute value and it does it by adding some extra controls on the parity of variable a that have no effect on the result of computation[1]. The semantics of program P is:

$$\begin{aligned}
[\![P]\!] \;=\; & \{\langle B_1:\bot\rangle\langle B_2:v_1\rangle\langle B_4:2*v_1+1\rangle\langle B_6:2*v_1+1\rangle\langle B_7:v_1\rangle \mid v_1\geq 0\}\cup\\
& \{\langle B_1:\bot\rangle\langle B_3:v_1\rangle\langle B_4:2*v_1\rangle\langle B_5:2*v_1\rangle\langle B_7:-v_1\rangle \mid v_1< 0\}
\end{aligned}$$

where $\langle B_i, val\rangle$ denotes the program state specifying the value val of variable a when entering block B_i and $\bot$ denotes the undefined value. Assume that we are interested in the analysis on the abstract domain *Sign* depicted on the left of Figure 2. The $Sign = \{\bot, +_0, -, \top\}$ abstract domain observes the sign of integer values and it is possible to define a GC between $\wp(\mathbb{Z})$ and *Sign* where the abstract element $+_0$ represents all positive values plus 0, the abstract element $-$ represents all negative values, while $\top$ represents all integer values and $\bot$ the emptyset. We denote with $[\![P]\!]^{Sign} \in \wp(\Sigma^*)$ the abstract interpretation of program P on the domain of *Sign*, where the values of variable a are interpreted on *Sign*.

$$\begin{aligned}
[\![P]\!]^{Sign} \;=\; & \{\langle B_1:\bot\rangle\langle B_2:+_0\rangle\langle B_4:+_0\rangle\langle B_6:+_0\rangle\langle B_7:+_0\rangle,\\
& \langle B_1:\bot\rangle\langle B_2:+_0\rangle\langle B_4:+_0\rangle\langle B_5:+_0\rangle\langle B_7,-\rangle[\textit{false positive}]\\
& \langle B_1:\bot\rangle\langle B_3:-\rangle\langle B_4:-\rangle\langle B_5:-\rangle,\langle B_7,+_0\rangle\\
& \langle B_1:\bot\rangle\langle B_3:-\rangle\langle B_4:-\rangle\langle B_6:-\rangle\langle B_7,-\rangle[\textit{false positive}]\}
\end{aligned}$$

Each abstract trace corresponds to infinitely many concrete traces. So for example the abstract trace $\langle B_1:\bot\rangle\langle B_2:+_0\rangle\langle B_4:+_0\rangle\langle B_6:+_0\rangle\langle B_7:+_0\rangle$ corresponds to the infinte set of concrete traces: $\{\langle B_1:\bot\rangle\langle B_2:v_1\rangle\langle B_4:v_2\rangle\langle B_6:v_3\rangle\langle B_7:v_3\rangle \mid v_1,v_2,v_3,v_4\geq 0\}$. Observe that the second and fourth abstract traces are false positives that the abstract analysis has to consider but that cannot happen during computation. This is because the guard at B_4

[1] The notation $\lfloor a/2\rfloor$ refers to the integer division that rounds the non-integer results towards the lower integer value.

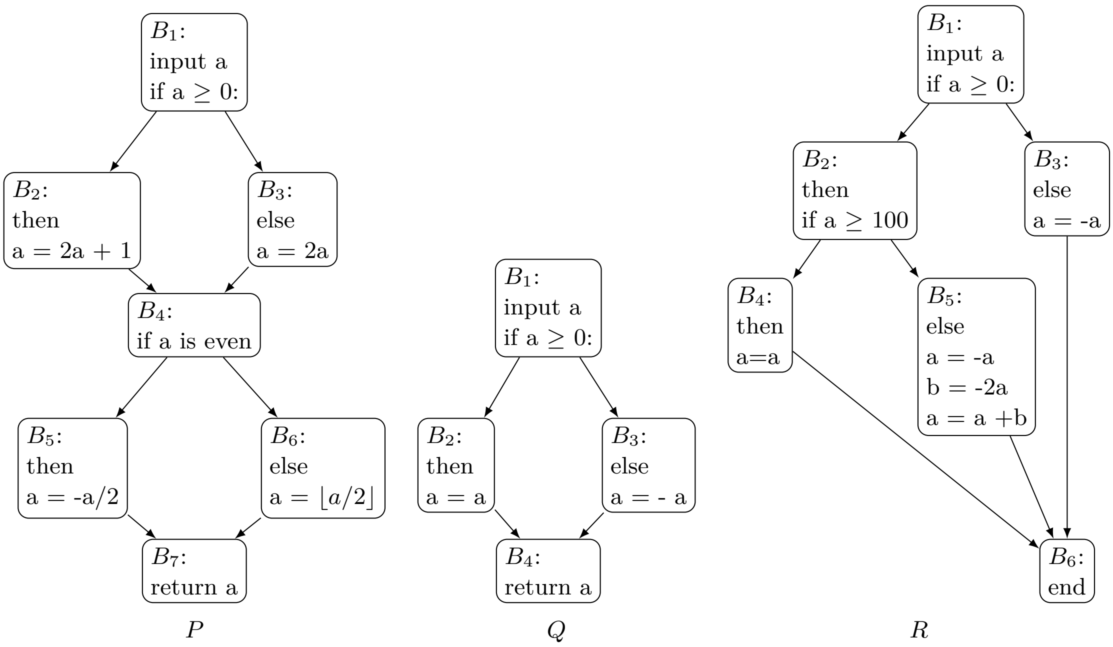

Figure 3 P, Q and R are functionally equivalent programs

cannot be precisely evaluated on *Sign* and therefore both branches are seen as possible. This happens because the abstract domain of *Sign* is not complete for the analysis of program P and we have $[\![P]\!] \subset [\![P]\!]^{Sign}$. This induces imprecision in the analysis on the abstract domain *Sign* that it is not able to conclude that the value of variable a is always positive at the end of execution. Let us denote with $[\![P]\!](B_i)$ and with $[\![P]\!]^{Sign}(B_i)$ the possible values that can be assumed by variable a at block B_i when reasoning on the concrete and abstract semantics respectively. In this case we have that $Sign([\![P]\!](B_7)) = Sign(\{v \mid v \geq 0\}) = +_0$ and this is more precise than $[\![P]\!]^{Sign}(B_7) = \sqcup_{Sign}\{+_0, -\} = \top$.

Transforming properties towards completeness

It is well known that completeness is a domain property and that abstract domains can be refined in order to become complete for the analysis of a given program [23]. The idea is that in order to make the analysis complete we need to add to the abstract domain those elements that are necessary to reach completeness. In this case, if we consider the abstract domain that observes the sign and parity of integer values we reach completeness. Thus, let us consider the domain $Sign \sqcap Parity$ depicted on the right of Figure 2, where *even* represents all the even integer values and *odd* represents all the odd integer values.

$$
\begin{aligned}
[\![P]\!]^{Sign \sqcap Parity} \quad = \quad & \{\langle B_1 : \bot\rangle\langle B_2 : (+_0, \mathit{even})\rangle\langle B_4 : (+_0, \mathit{odd})\rangle\langle B_6 : (+_0, \mathit{odd})\rangle\langle B_7 : +_0\rangle \\
& \langle B_1 : (+_0, \bot)\rangle\langle B_2 : (+_0, \mathit{odd})\rangle\langle B_4 : (+_0, \mathit{odd})\rangle\langle B_6 : (+_0, \mathit{odd})\rangle\langle B_7 : +_0\rangle \\
& \langle B_1 : (-, \bot)\rangle\langle B_3 : (-, \mathit{even})\rangle\langle B_4 : (-, \mathit{even})\rangle\langle B_5 : (-, \mathit{even})\rangle\langle B_7 : +_0\rangle \\
& \langle B_1 : (-, \bot)\rangle\langle B_3 : (-, \mathit{odd})\rangle\langle B_4 : (-, \mathit{even})\rangle\langle B_5 : (-, \mathit{even})\rangle\langle B_7 : +_0\rangle\}
\end{aligned}
$$

As we can see all the abstract traces are able to precisely observe that variable a is positive at the end of the execution and that it can be either even or odd. Indeed, we have completeness with respect to the $Sign \sqcap Parity$ property $Sign \sqcap Parity([\![P]\!](B_7)) = [\![P]\!]^{Sign \sqcap Parity}(B_7) = +_0$.

Thus, a possible way for tuning the precision of static analysis is to transform the property that we want to analyse in order to reach completeness, there exists a systematic methodology that allows us to add the minimal amount of elements to the abstract domain in order to make the analysis complete for a given program [23].

Transforming programs towards completeness

The way in which programs are written affects the precision of the analysis. For example we can easily write a program functionally equivalent to P but for which the analysis on *Sign* is complete. Consider, for example, program Q as the one in the middle of Figure 3, we have that:

$$[\![Q]\!] = \{\langle B_1 : \bot\rangle\langle B_2 : v\rangle\langle B_4 : v\rangle \mid v \geq 0\} \cup \{\langle B_1 : \bot\rangle\langle B_3 : v\rangle\langle B_4 : -v\rangle \mid v < 0\}$$

$$[\![Q]\!]^{Sign} = \{\langle B_1 : \bot\rangle\langle B_2 : +_0\rangle\langle B_4 : +_0\rangle, \langle B_1 : \bot\rangle\langle B_3 : -\rangle\langle B_4 : +_0\rangle\}$$

This makes it clear how the abstract computation loses information regarding the modulo of the value of variable a, while it precisely observes the positive value of a at the end of execution. Indeed, in this case we have that: $Sign([\![Q]\!](B_7)) = Sign(\{v \mid v \geq 0\}) = +_0 = [\![Q]\!]^{Sign}(B_7)$. It is worth studying the possibility of transforming programs in order to make a certain analysis complete. In a recent work [6] the authors introduced the notions of complete clique $\mathbb{C}(P, A)$ and incomplete clique $\bar{\mathbb{C}}(P, A)$ that represent the set of all programs that are functionally equivalent to P and for which the analysis on the abstract domain A is respectively complete and incomplete. They prove that there are infinitely many abstractions for which the systematic removal of false positives for all programs is impossible. Moreover, they observe that false positives are related to the evaluation of boolean predicates that the abstract domain is not able to evaluate precisely (as we have seen in our earlier example). The authors claim that their investigation together with the poof system in [24] should be used as a starting point to reason on a code refactoring strategy that aims at modifying a given program in order to gain precision with respect to a predefined analysis. Given an abstract domain A, the final goal would be to derive a program transformation $\mathcal{T}_A : Prog \to Prog$ that given a program $P \in \bar{\mathbb{C}}(P, \mathcal{A})$ for which the analysis $\mathcal{A}$ is incomplete, namely $\mathcal{A}([\![P]\!]) \neq [\![P]\!]^{\mathcal{A}}$, transforms it into a program $\mathcal{T}(P) \in \mathbb{C}(P, \mathcal{A})$ for which the analysis is complete, namely $\mathcal{A}([\![P]\!]) = [\![P]\!]^{\mathcal{A}}$.

These recent promising works suggest how to proceed in the investigation of program transformations as a mean for gaining precision in static program analysis.

3.2 Dynamic Analysis

Testing is typically used to discover failures (or bugs), namely an incorrect program behaviour with respect to the requirements or the description of the expected program behaviour. Precision in program testing is expressed in terms of soundness: the ideal situation where no false negatives are present. When speaking of failures, this happens when the executions considered in the test set exhibit at least one behaviour for each one of the failures present in the program. Indeed, when this happens, testing allows us to detect all the failures in the program. It is clear that the choice of the input set to use for testing is fundamental in order to minimise the number of false negatives. What we have just said holds when testing aims at detecting failures as well as for the analysis of any property of traces (as for example the order in which memory cells are accessed, the target of jumps, etc.). Let us denote with $\mathbb{I}_P$ the input space of the possible input values needed to complete an execution of program

P under testing[2]. Dynamic analysis considers a finite subset of the input space, called the input set $InSet \subseteq_F \mathbb{I}_P$, that identifies the input values that are used for execution. The execution traces generated by the input set define the test set, which is the finite set of traces used by dynamic analysis to reason on program behaviour. Given an input value $x \in \mathbb{I}_P$ we denote with $P(x) \in [\![P]\!]$ the execution of program P when fed with input x.

As argued above, the main source of imprecision in testing is that the number of potential inputs for most programs is so large as to be effectively infinite. Since we cannot test with all inputs, researchers typically recur to the use of coverage criteria in order to decide which test inputs to use. A coverage criterion C induces a partition on the input space and in order to minimise the false negatives the input set should contain at least one element for each class of the partition. In the left part of Figure 4 we consider a typical coverage criterion, called path coverage, for the testing of program Q in Figure 3. Path coverage criterion is satisfied when for each path in the control flow graph of the program there exists at least one execution in the test set that follows that path. When considering program Q it is immediate to derive from the coverage criterion the partition of the input space: the class of positive integer values (that follow the path $B_1 \to B_2 \to B_4$) and the class of negatives integer values (that follow the path $B_1 \to B_3 \to B_4$). In this case the coverage criterion is satisfied by every input set that contains at least one positive integer value and one negative integer value.

Since it is the coverage criterion that determines the input set and therefore the executions that are considered by the dynamic analysis, it is very important to select a *good* coverage criterion. However, it is not clearly stated or formally defined what makes a coverage criterion good [1], and this may be one of the reasons why many coverage criteria have been developed by researchers. Generally speaking, there are some features that it is important to consider when speaking of coverage criterion such as:

- the difficulty of deriving the rules to partition the input space with respect to the coverage criterion;
- the difficulty of generating an input set that satisfies the coverage criterion, namely that contains at least one input for each one of the classes in which the input space has been partitioned;
- how well a test set that satisfies the coverage criterion guarantees the absence of false negatives.

To the best of our knowledge there is no general framework that formalises the relation between coverage criterion, partition of the input space and false negatives in the dynamic analysis of a semantic program property. Indeed, while the soundness of dynamic analysis may not be possible in general, we think that it would be interesting to study the soundness of dynamic analysis of a program with respect to a specific semantic property (as usually done when reasoning about completeness in static analysis). We believe that this formal investigation would help in better understanding the cause of false negatives and would be useful in reducing them.

3.2.1 Towards a formal framework for dynamic analysis

We formalise the splitting of the input space induced by a coverage criterion C in terms of an equivalence relation $\mathcal{R}_I^C \in Eq(\mathbb{I}_P)$, and this allows us to formally define when an input set satisfies a coverage criterion.

[2] In this work, for simplicity but with no loss of generality, we speak of input values while in the general case we may need collections of values in order to complete an execution of the software under test.

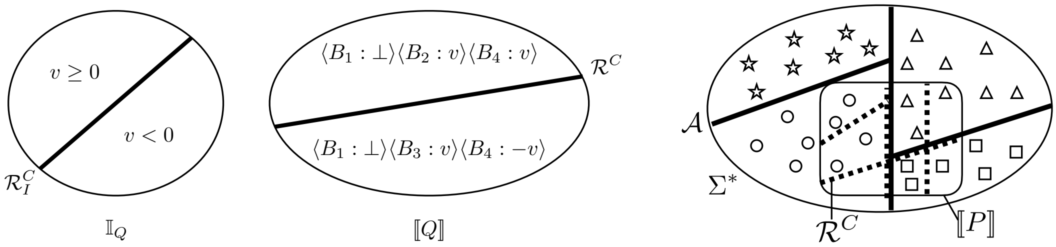

Figure 4 Path coverage criterion on program Q of Figure 3, and soundness conditions

▶ **Definition 1.** *Given a program P, an input set $InSet \subseteq_F \mathbb{I}_P$ and a coverage criterion C, we say that $InSet$ satisfies C, denoted $InSet \models C$, iff: $\forall [x]_{\mathcal{R}^C} \in \mathbb{I}_P/\mathcal{R}_I^C$ we have that $InSet \cap [x]_{\mathcal{R}_I^C} \neq \emptyset$.*

We have seen this in Figure 4 when considering the partition induced in the input space of program Q and observing that an input set satisfies the path coverage criterion when it contains at least one positive and one negative integer value. When considering coverage criteria we need to take into account infeasible requirements: for example when considering coverage criteria related to the paths of the control flow graph we have to handle infeasible paths as it is not possible to define input values that follow these paths (as for example paths $B_1 \to B_2 \to B_4 \to B_5 \to B_7$ and $B_1 \to B_3 \to B_4 \to B_5 \to B_7$ of program P). This is a known challenging problem in dynamic analysis and testing as the detection of infeasible test requirements is undecidable for most coverage criteria [1]. This means that some preliminary analysis is needed in order to ensure the feasibility of the coverage criteria, namely to ensure that it is possible to generate an input set that satisfies a given coverage criterion. Otherwise, we need to somehow quantify how much the input set satisfies the coverage criterion, for example considering the percentage of equivalence classes that are covered by the input set. In this work we do not address this problem and we assume the feasibility of the coverage criteria.

Observe that the equivalence relation $\mathcal{R}_I^C \in Eq(\mathbb{I}_P)$ naturally induces an equivalence relation on traces $\mathcal{R}^C \in Eq([\![P]\!])$ where $(\sigma_1, \sigma_2) \in \mathcal{R}^C$ iff $\exists x_1, x_2 \in \mathbb{I}_P : P(x_1) = \sigma_1$, $P(x_2) = \sigma_2$ and $(x_1, x_2) \in \mathcal{R}_I^C$. Thus, we can say that a given coverage criterion, and therefore any test set that satisfies that coverage criterion, can be associated to a partition of program trace semantics. Our idea is that the partition of the program trace semantics induced by the coverage criterion could be used to reason on the class of semantic program properties for which the coverage criterion can ensure soundness. To this end, we need to represent semantic program properties in a way that can be compared with partitions on traces.

Properties of traces are naturally modelled as abstract domains, namely as closure operators in $uco(\wp(\Sigma^*))$. A semantic property $\rho \in uco(\wp(\Sigma^*))$ maps an execution trace (or a set of execution traces) to the minimal set of traces that cannot be distinguished by the considered property. Each closure operator $\rho \in uco(\wp(\Sigma^*))$ induces an equivalence relation $\mathcal{R}^\rho \in Eq(\Sigma^*)$: $\sigma_1 \mathcal{R}^\rho \sigma_2$ iff $\rho(\{\sigma_1\}) = \rho(\{\sigma_2\})$, where traces are grouped together if they are mapped in the same element by abstraction ρ. In the following, we model the properties of traces as equivalence relations over traces or equivalently as partitioning closures in $uco(\wp(\Sigma^*))$, and we denote these properties as $\mathcal{A} \in Eq(\Sigma^*)$. According to [29] there is more than one closure that maps to the same equivalence relations, thus considering the partitions induced by closure operators corresponds to focusing on the set of partitioning closures (which is a proper subset of closure operators over $\wp(\Sigma^*)$). This allows us to express properties of the single traces but not relational properties that have to take into account more than one trace. This means that we can use equivalence relations in $Eq(\Sigma^*)$ to express properties such as: the order of successive accesses to memory, the order of execution of instructions, the location of the first instruction of a function, the target of jumps, function location, possible data values at certain program points, the presence of a bad states in the trace, and so on. These are properties of practical interest in dynamic program analysis. What we cannot express are properties on sets of traces, the so called hyper-properties, that express relational properties among traces, like non-interference. The extension of the framework to closures that are not partitioning is left as future work. This allows us to formally model the soundness of dynamic analysis.

▶ **Definition 2.** *Given a program P and a property $\mathcal{A} \in Eq(\Sigma^*)$, the dynamic analysis $\mathcal{A}$ on input set $InSet \subseteq_F \mathbb{I}_P$ is sound, denoted $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$, if $\forall[\sigma]_{\mathcal{A}} \in [\![P]\!]/\mathcal{A}$ we have that $[\sigma]_{\mathcal{A}} \cap InSet \neq \emptyset$.*

This precisely captures the fact that dynamic analysis needs to observe the different behaviours of the program with respect to the property of interest in order to be sound. Indeed, when considering a program P and a property $\mathcal{A}$ it is enough to observe a single trace in an equivalence class $[\sigma]_{\mathcal{A}} \subseteq [\![P]\!]$ in order to observe how property $\mathcal{A}$ behaves in all the traces of program P that belong to that equivalence class. If we consider program Q in Figure 3 we have that in order to precisely observe the evolution of the sign property along the execution we have to consider at least one trace that follows the path $B_1 \to B_2 \to B_4$ and one trace that follows the path $B_1 \to B_3 \to B_4$ as depicted in Figure 4.

Modelling program properties as equivalence relations makes it easy to compare them with the coverage criteria and to reason on soundness.

▶ **Theorem 3.** *Given a program P, a coverage criterion C, an input set $InSet \subseteq_F \mathbb{I}_P$ and a property $\mathcal{A} \in Eq(\Sigma^*)$, we have that if $\mathcal{R}^C \preceq \mathcal{R}^{\mathcal{A}}_{|[\![P]\!]}$ and $InSet \models C$, then $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$.*

Proof. $InSet \models C$ therefore $\forall[x]_{\mathcal{R}^C} \in \mathbb{I}_P/\mathcal{R}^C$ we have that $InSet \cap [x]_{\mathcal{R}^C} \neq \emptyset$. Since $\mathcal{R}^C \preceq \mathcal{A}_{|[\![P]\!]}$ we have that for each equivalence class $[\sigma]_{\mathcal{R}^C}$ there exists an equivalence class $[\sigma]_{\mathcal{A}_{|[\![P]\!]}}$ that $[\sigma]_{\mathcal{R}^C} \subseteq [\sigma]_{\mathcal{A}_{|[\![P]\!]}}$. This implies that for every $[\sigma]_{\mathcal{A}_{|[\![P]\!]}} \in [\![P]\!]/\mathcal{A}$ we have that $[\sigma]_{\mathcal{A}_{|[\![P]\!]}} \cap InSet \neq \emptyset$ and therefore $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$. ◀

In Figure 4 on the right we provide a graphical representation of the above theorem. Traces in Σ^* exhibit different attributes with respect to property $\mathcal{A}$ and this is represented by the different shapes: circle, triangle, square and star. Trace partition is then represented by the thick lines that group together traces that are undistinguishable with respect to property $\mathcal{A}$. Dotted lines are used to represent a trace partition induced by coverage criterion C on the traces of P and that ensures the absence of false negatives in the analysis. Indeed, from the graphical representation it is clear that when $InSet \models C$ then $InSet$ contains at least a trace for each equivalence class of $\mathcal{R}^C$, and this implies that it contains at least a trace for each one of the possible attributes (circle, triangle and square) that traces in $[\![P]\!]$ can exhibit with respect to property $\mathcal{A}$. This allows us to characterise the set of properties for which a given coverage criterion can ensure soundness.

▶ **Definition 4.** *Given a coverage criterion C on a program P, we define the set of properties $\Pi(C) \overset{def}{=} \{\mathcal{A} \in Eq(\Sigma^*) \mid \mathcal{R}^C \preceq \mathcal{A}_{|[\![P]\!]}\}$ that are coarsest than the equivalence relation induced by the coverage criterion.*

It follows that any input set that satisfies a coverage criterion C on a program P would lead to a sound dynamic analysis on any property in $\Pi(C)$.

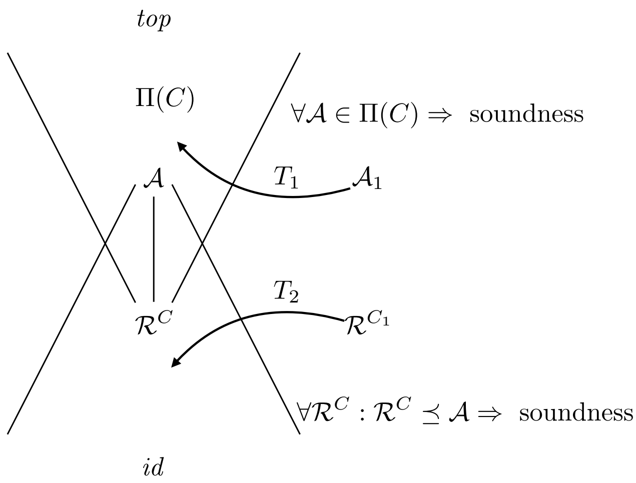

Figure 5 Comparing $\mathcal{R}^C$ and $\mathcal{A}$ for soundness

▶ **Corollary 5.** *Given a coverage criterion C on a program P, and input set $InSet \subseteq_F \mathbb{I}_P$ such that $InSet \models C$, than $\forall \mathcal{A} \in \Pi(C)$ we have that $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$.*

In Figure 5 we summarise the relation between coverage criteria and soundness of a particular program property. Given a program P, Figure 5 depicts the domain of equivalence relations over $[\![P]\!]$ where *id* denotes the most fine equivalence relation that corresponds to the identity relation, $\forall \sigma_1, \sigma_2 \in [\![P]\!] : (\sigma_1, \sigma_2) \in id$ iff $\sigma_1 = \sigma_2$, and *top* denotes the coarser equivalence relation that sees every trace as equivalent $\forall \sigma_1, \sigma_2 \in [\![P]\!]$ it holds that $(\sigma_1, \sigma_2) \in top$. As stated in Theorem 3 whenever $\mathcal{R}^C \preceq \mathcal{A}_{|[\![P]\!]}$ then the coverage criterion C can be used to ensure soundness of the analysis of property $\mathcal{A}$ on program P. As stated by Corollary 5 a coverage criterion C can ensure soundness for all those properties in $\Pi(C)$.

Following our reasoning, the most natural coverage criterion for a given semantic property $\mathcal{A}$ is the one for which $\mathcal{R}^C = \mathcal{A}$, namely the coverage criterion whose partition on states corresponds to the property under analysis. In the literature there exists many different coverage criteria and some of them turn out to be equivalent when compared with respect to the partition that they induce on the input space. It has been observed that all existing test coverage criteria can be formalised on four mathematical structures: input domains, graphs, logic expressions, and syntax descriptions (grammars) [1]. Even if these coverage criteria are not explicitly related to the properties being analysed they have probably been designed while having in mind the kind of properties of interest. For example, some of the most widely known coverage criteria are based on graph features and are typically used for the analysis of properties related to a graphical representation of programs, like control flow or data flow properties of code or variables that can be verified on the control flow graph of a program, or function calls that can be verified on the call graph or a program, and so on. For example code coverage requires the execution of all the basic blocks of a control flow graph and wants to ensure that all the reachable instructions of a program are considered at least in one execution of the test set.

What we have stated so far allows us to begin to answer the question regarding how well the coverage criterion behaves with respect to the analysis of a given semantic property (when this can be modelled as a partitioning closure on the powerset of program traces). The design of an automatic or systematic strategy for the generation of an input set that covers a given coverage criterion remains an open challenge that deserves further investigation.

Transforming properties towards soundness

There are two questions that naturally arise from our reasoning and that would be interesting to investigate regarding the systematic transformation of the property under analysis or the coverage criterion towards soundness.

1. Consider a program P, a coverage criterion C that induces a partition $\mathcal{R}^C \in Eq(\llbracket P \rrbracket)$ on the traces of program P and a trace property $\mathcal{A}_1$ for which the coverage criterion C cannot ensure soundness. We wonder if it is possible to design a systematic transformation of property $\mathcal{A}_1$ that, by grouping some of its equivalence classes, returns a trace property for which we have soundness when C is satisfied by the input set. It would be interesting to understand when this transformation is possible without reaching *top*, i.e., while still being able to distinguish trace properties. This is depicted by the arrow labeled with T_1 in the upper part of Figure 5.
2. Consider a program P, a coverage criterion C_1 that induces a partition $\mathcal{R}^{C_1} \in Eq(\llbracket P \rrbracket)$ on the traces of program P and a trace property $\mathcal{A}$ for which the coverage criterion C_1 cannot ensure soundness. We wonder if it is possible to design a systematic transformation of $\mathcal{R}^{C_1}$ that, by further splitting its equivalence classes, returns a partition of the program traces, and therefore a coverage criterion, that when satisfied by the input set ensures soundness for the analysis of property $\mathcal{A}$. In this case it is interesting to investigate when this refinement is possible without ending up with the identity relation, namely without collapsing to *id* where all program traces needs to be considered for coverage. This is depicted by the arrow labeled with T_2 in the bottom part of Figure 5.

Transforming programs towards soundness

As for static analysis also for dynamic analysis the way in which programs are written influences the precision of the analysis either because they expand the input set that satisfies a given coverage criterion, thus requiring the observation of more program runs, or because they complicate the automatic/systematic extraction of an input set that satisfies a given coverage criterion. We focus on the first case since we still have to formally investigate the extraction of input sets for a given coverage criterion, namely the input generation and input recogniser procedure.

Let us consider program R on the right of Figure 3 that computes the absolute value of an integer value and does it by adding some extra control on the range of the input integer value in order to proceed with the computation of the modulo in some syntactically different, but semantically equivalent ways. Indeed, in this example it is easy to observe that blocks B_4 and B_5 are equivalent, but we can think about more sophisticated ways to write equivalent code in such a way that it would be difficult for the analyst to automatically recognise that they are equivalent. If we consider again the path coverage criterion we can observe that in order to cover the control flow graph of program R we need at least three input values: a negative integer, a positive integer smaller than 100 and a positive integer greater than or equal to 100. Of course what is done in block B_2 can be replicated many times, as far as we are able to write blocks that are syntactically different but semantically equivalent to B_4 or B_3. According to our framework, path coverage is more complicated to reach on program R than on program Q. Indeed, in this case, every input set that satisfies path coverage for program R also satisfies path coverage for program Q while the converse does not hold in general. This reasoning is limited to the amount of traces that we need to satisfy a given coverage criterion and does not take into account the difficulty of generating such traces. Of course both aspects would need to be taken into account by our formal framework.

Moreover, as done for static analysis in [6], it would be interesting to define the notions of sound clique $\mathbb{S}(P, InSet, \mathcal{A})$ and of unsound clique $\bar{\mathbb{S}}(P, InSet, \mathcal{A})$ that represent the sets of all programs that are functionally equivalent to P and for which the dynamic analysis of property $\mathcal{A}$ on input set $InSet \subseteq \mathbb{I}_P$ is respectively sound and not sound:

$$\mathbb{S}(P, InSet, \mathcal{A}) \stackrel{\text{def}}{=} \{Q \in \mathit{Prog} \mid \mathit{Den}(\llbracket P \rrbracket) = \mathit{Den}(\llbracket Q \rrbracket), InSet \stackrel{s}{\rightsquigarrow} \mathcal{A}(P)\}$$

$$\bar{\mathbb{S}}(P, InSet, \mathcal{A}) \stackrel{\text{def}}{=} \{Q \in \mathit{Prog} \mid \mathit{Den}(\llbracket P \rrbracket) = \mathit{Den}(\llbracket Q \rrbracket), Q \notin \mathbb{S}(P, InSet, \mathcal{A})\}$$

We plan to study the existence of transformations from $\bar{\mathbb{S}}(P, InSet, \mathcal{A})$ to $\mathbb{S}(P, InSet, \mathcal{A})$ in order to rewrite a program toward soundness. It is interesting to understand which are the properties for which this can be done in a systematic way and what is the key for reaching soundness. The intuition is that for reaching soundness with respect to a property $\mathcal{A}$ on an input set $InSet$ we should choose programs whose variations of property $\mathcal{A}$ are all considered by the input set as stated in Theorem 3. Thus, in general, if we reduce variations of the considered property by merging traces that are functionally equivalent even if they have diversified $\mathcal{A}$ properties we would probably facilitate soundness. This needs to be formally understood, proved and validated on some existing dynamic analysis.

4 Software protection: a new perspective

In the software protection scenario we are interested in preventing program analysis while preserving the intended behaviour of programs. To face this problem Collberg et al. [9] introduced the notion of code obfuscation: program transformations designed with the explicit intent of complicating and degrading program analysis while preserving program functionality. Few years later Barak et al. [3] proved that it is not possible to obfuscate everything but the input-output behaviour for all programs with limited penalty in performances. However, it is possible to relax some of the requirements of Barak et al. and design obfuscating techniques that are able to complicate certain analysis of programs. This is witnessed by the great amount of obfuscation tools and techniques that researchers, both from academia and industry, have been developing in the last twenty years [8]. What it means for a program transformation to complicate program analysis is something that needs to be formally stated and proved when defining new obfuscating transformations. The extent to which an obfuscating technique complicates, and therefore protects, the analysis of certain program properties is referred to as *potency* of the obfuscation. A formal proof of the quality of obfuscation in terms of its potency is very important in order to compare the efficiency of different obfuscation techniques and in order to understand the degree of protection that they guarantee. Unfortunately, a unifying methodology for the quantitative evaluation of software protection techniques is still an open challenge, as witnessed by the recent Dagstuhl Seminar on this topic [20]. What we have are specific measurements done when new techniques are proposed, or formal proofs that reduce the analysis of obfuscated programs to well known complex analysis tasks (like alias analysis, shape analysis, etc.).

In our framework, complicating program analysis means inducing imprecision in the results of the analysis of the obfuscated program with respect to the results of the analysis of the original program. This means that code obfuscation should induce false positives in static program analysis and false negatives in dynamic program analysis.

4.1 Program transformations against static program analysis

The abstract interpretation framework has been used to reason on the semantic properties that code obfuscation transformations are able to protect and the ones that they can still be analysed on the obfuscated program. It has been observed that a program property expressed by an abstract domain $\mathcal{A}$ is obfuscated (protected) by an obfuscation $\mathcal{O} : \mathit{Prog} \to \mathit{Prog}$ on a program P whenever $[\![P]\!]^{\mathcal{A}} \leq_{\mathcal{A}} [\![\mathcal{O}(P)]\!]^{\mathcal{A}}$, namely when the analysis $\mathcal{A}$ on the obfuscated program returns a less precise result with respect to the analysis of the same property on the original program P. The spurious information added to the analysis by the obfuscation is the noise that confuses the analyst, thus making the analysis more complicated. The relation between potency of code obfuscation and the notion of (in)completeness in abstract interpretation has been proven, as obfuscating a property means to induce incompleteness in its analysis [22]. So, for example, the insertion of a true opaque predicate O^T (see the program in the middle of Figure 1) would confuse all those analyses that are not able to precisely evaluate such a predicate and have to consider both branches as possible. No confusion is added for those analyses that are able to precisely evaluate the opaque predicate and consider only the true branch as possible, namely those analyses that are complete for the evaluation of the predicate value. Following this idea, a formal framework based on program semantics and abstract interpretation has been developed, where it is possible to formally prove that a property is obfuscated by a given program transformation, compare the efficiency of different obfuscating techniques in protecting a given property, define a systematic strategy for the design of a code obfuscation technique for protecting a given program property [17, 19, 22, 25]. This semantic understanding of the effects that code obfuscation has on the semantics and semantic properties of programs as shown its usefulness also in the malware detection scenario where malware writers use code obfuscation to evade automatic detection [15, 16].

Thus we can say that the effects of functionality preserving program transformations on program semantics and on the precision of the results of static analysis has been extensively studied and a mature formal framework has been provided [15, 16, 17, 19, 22, 25].

4.2 Program transformations against dynamic program analysis

To the best of our knowledge, the effects of functionality preserving program transformations on the precision of dynamic analysis have not been fully investigated yet. Following our reasoning, the general idea is that dynamic analysis is complicated by program transformations that induce false negatives while preserving program's functionality. Let $\mathcal{A} \in \mathit{Eq}(\Sigma^*)$ denote a property of interest for dynamic analysis. Inducing false negatives for the analysis of a property $\mathcal{A}$ can be done by exploiting the partial observation of program's executions innate in the test set, and thus adding traces that do not belong to the test set and have a different $\mathcal{A}$ property. Thus, the key for software protection against dynamic analysis is software *diversification* with respect to the property under analysis. The ideal obfuscation against the dynamic analysis of property $\mathcal{A}$ should specialise programs with respect to every input in such a way that every input exhibits a different behaviour for property $\mathcal{A}$. Namely, an ideal obfuscation against $\mathcal{A}$ is a program transformation $\mathcal{O} : \mathit{Prog} \to \mathit{Prog}$ such that $\forall \sigma_1, \sigma_2 \in [\![\mathcal{O}(P)]\!]$ we have that $\mathcal{A}(\sigma_1) = \mathcal{A}(\sigma_2) \Leftrightarrow \sigma_1 = \sigma_2$. In this ideal situation in order to avoid false negatives the analyst should consider every possible execution trace of $\mathcal{O}(P)$ since each trace exhibits a different aspects of property $\mathcal{A}$, so missing a trace would mean to miss such an aspect. This intuition is confirmed in a preliminary work in this direction where it is shown how diversification is the basis of existing software protection techniques against dynamic analysis [18]. This work provides a topological characterisation of the soundness of the dynamic analysis of properties expressed as equivalence relations (as we have done in Section 3.2.1). This formal characterisation is then used to define the notion of transformation potency for dynamic analysis.

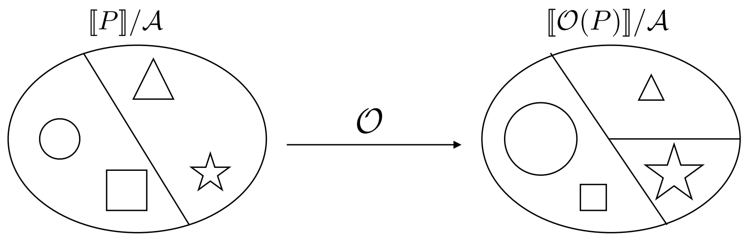

Figure 6 Transformation Potency

▶ **Definition 6.** *A functionality preserving program transformation $\mathcal{O}$: Prog $\rightarrow$ Prog is potent for the analysis of $\mathcal{A} \in Eq(\Sigma^*)$ of program P if:*

- $\forall \sigma_1, \sigma_2 \in [\![\mathcal{O}(P)]\!] : [\sigma_1]_\mathcal{A} = [\sigma_2]_\mathcal{A}$, $\forall \nu_1, \nu_2 \in [\![P]\!] : Den(\nu_1) = Den(\sigma_1), Den(\nu_2) = Den(\sigma_2)$ *then* $[\nu_1]_\mathcal{A} = [\nu_2]_\mathcal{A}$
- $\exists \nu_1, \nu_2 \in [\![P]\!] : [\nu_1]_\mathcal{A} = [\nu_2]_\mathcal{A}$ *for which* $\exists \sigma_1, \sigma_2 \in [\![\mathcal{O}(P)]\!] : Den(\nu_1) = Den(\sigma_1), Den(\nu_2) = Den(\sigma_2)$ *such that* $[\sigma_1]_\mathcal{A} \neq [\sigma_2]_\mathcal{A}$

Figure 6 provides a graphical representation of the notion of potency. On the left we have the traces of the original program P partitioned according to the equivalence relation $\mathcal{A}$ induced by the property of interest, while on the right we have the traces of the transformed program $\mathcal{O}(P)$ partitioned according to $\mathcal{A}$. Traces that are denotationally equivalent have the same shape (triangle, square, circle, oval), but different dimension since they are in general different traces. The first condition means that the traces of $\mathcal{O}(P)$ that property $\mathcal{A}$ maps to the same equivalence class (circle and square), are denotationally equivalent to traces of P that property $\mathcal{A}$ maps to the same equivalence class. This means that what is grouped together by $\mathcal{A}$ on $[\![\mathcal{O}(P)]\!]$ was grouped together by $\mathcal{A}$ on $[\![P]\!]$, modulo the denotational equivalence of traces. The second condition requires that there are traces of P (triangle and star) that property $\mathcal{A}$ maps to the same equivalence class and whose denotationally equivalent traces in $\mathcal{O}(P)$ are mapped by $\mathcal{A}$ to different equivalence classes. This means that a defense technique against dynamic analysis with respect to a property $\mathcal{A}$ is successful when it transforms a program into a functionally equivalent one for which property $\mathcal{A}$ is more diversified among execution traces. This implies that it is necessary to collect more execution traces in order for the analysis to be precise. At the limit we have an optimal defense technique when $\mathcal{A}$ varies at every execution trace.

The above definition of transformation potency for dynamic analysis has been validated by modelling in the proposed framework some existing software defence strategies against dynamic analysis for the extraction of the control flow graph of programs like Range Dividers [2] and Gadget diversification [30]. In both cases it is possible to show that the proposed transformations complicate the dynamic extraction of the control flow graph by adding new diversified paths to the control flow graph, as stated in Definition 6. In the following we report a simple example from [18] that shows how the key for obfuscating properties of data values for dynamic analysis is diversification.

▶ **Example 7.** Consider the following programs P and Q that compute the sum of natural numbers from $x \geq 0$ to 49 (we assume that the inputs values for x are natural numbers).

```
P
input x;
sum := 0;
while x < 50
• ≀ X = [0, 49] ≀
  sum := sum + x;
  x := x + 1;
```

```
Q
input x;
n : = select(N,x)
x := x * n;
sum := 0;
while x < 50 * n
• ≀ X = [0, n * 50 − 1] ≀
  sum := sum + x/n;
  x := x + n;
x := x/n;
```

Consider a dynamic analysis that observes the maximal value assumed by x at program point $\bullet$. For every possible execution of program P we have that the maximal value assumed by x at program point $\bullet$ is 49. Consider a state $s \in \Sigma$ as a tuple $\langle pp, [val_x, val_{sum}]\rangle$, where pp denotes the current program point, val_x and val_{sum} denote the current values of variables x and sum respectively. We define a function $\tau : \Sigma \to \mathbb{N}$ that observes the value assumed by x at state s when s refers to program point $\bullet$, and function $max : \Sigma^* \to \mathbb{N}$ that observes the maximal value assumed by x at $\bullet$ along an execution trace:

$$\tau(s) \stackrel{\text{def}}{=} \begin{cases} val_x & \text{if } pp = \bullet \\ \emptyset & \text{otherwise} \end{cases} \qquad max(\sigma) \stackrel{\text{def}}{=} max(\{\tau(s) \mid s \in \sigma\})$$

This allows us to define the equivalence relation $\mathcal{A}_{max} \in Eq(\Sigma^*)$ that observes traces with respect to the maximal value assumed by x at $\bullet$, as $(\sigma, \sigma') \in \mathcal{A}_{max}$ iff $max(\sigma) = max(\sigma')$. We can observe that all the execution traces of P belong to the same equivalence class of $\mathcal{A}_{max}$. In this case, a dynamic analysis of property $\mathcal{A}_{max}$ on P is sound whenever the test set contains at least one execution trace of P. This happens because the property that we are looking for is an invariant property of program executions and it can be observed on any execution trace.

Let us now consider program Q equivalent to P, i.e., $Den[\![P]\!] = Den[\![Q]\!]$, where the value of x is diversified by multiplying it by the parameter n. The guard and the body of the `while` are adjusted in order to preserve the functionality of the program. When observing property $\mathcal{A}_{max}$ on Q, we have that the maximal value assumed by x at program point $\bullet$ is determined by the parameter n generated in the considered trace. The statement `n:=select(N,x)` assigns to n a value in the range $[0, N]$ depending on the input value x. We have that the traces of program Q are grouped by $\mathcal{A}_{max}$ depending on the value assumed by n. Thus, $\mathcal{A}([\![Q]\!])$ contains an equivalence class for every possible value assumed by n during execution. This means that the transformation that rewrites P into Q is potent according to Definition 6. Dynamic analysis of property $\mathcal{A}_{max}$ on program Q is sound if the test set contains at least one execution trace for each of the possible values of n generated during execution.

5 Open research directions

We have provided an unifying view of the relations between properties and program transformations and the precision of static and dynamic analysis in the standard analysis scenario and in the software protection scenario. Researchers have proposed possible ways for tuning the precision of static analysis while less attention has been posed to the formal investigation of dynamic analysis. In this context it is worth to mention the recent work of O'Hearn [26] that defines a formalism called incorrectness logic, which is similar to Hoare's logic, and allows us to prove the presence of bugs but not their absence, thus capturing the essence of program testing. The incorrectness logic is based on a under-approximation triple that plays a dual role when compared to the standard over-approximation triple that we are used to see in Hoare's logic. Indeed, while logic and symbolic reasoning are useful since they can cover many states or program paths at once, they do not allow in general to cover all paths and this makes it difficult to prove the absence of errors. The author claims the necessity and usefulness of incorrectness logic that formalises under-approximate reasoning in order to provide a logical proof of the presence of bugs. Such reasoning should of course be combined with standard correctness proof in order to obtain a global view of program's runtime behaviour. The incorrectness logic of O'Hearn does not try to gain soundness, namely to avoid or reduce false negatives, but provides formal proofs for what can be derived in an unsound context. Our idea is to investigate the extent to which it is possible to induce or force soundness by modifying either the program, the property to be analysed or the coverage criterion. Once we have understood when and how soundness can be forced we should see how this interacts with incorrectness logic.

The preliminary work done in the investigation of program and properties transformations towards sound dynamic analysis have pointed out many interesting aspects that need to be studied and that we list below as future research directions.

The preliminary results that relate program properties, coverage criteria and the soundness of the analysis should be generalised and extended to properties that cannot be modelled as partitioning closures. Soundness of the analysis and transformation potency should be redefined probably in terms of join-irreducible elements instead of equivalence classes. This further investigation would probably lead to a classification of the properties usually considered by dynamic analysis based on the domain model needed to express them: properties of traces, properties of sets of traces, relational properties, hyper-properties. For each class of properties it would then be interesting to derive a suitable obfuscation strategy. This unifying framework would provide a common ground where to interpret and compare the potency of different software protection techniques in harming dynamic analysis.

As regarding the transformation of properties towards soundness, we plan to verify if and when it is possible to refine the coverage criterion C in order to ensure soundness with respect to a given property $\mathcal{A}$, or when it is possible to further abstract the semantic property $\mathcal{A}$ in order to make it sound for a given coverage criterion C. This should be done starting with properties that can be expressed as partitioning closures and then generalised to the other classes of properties.

As regarding the transformation of programs towards soundness, it is important to investigate when it is possible to transform a program P for which the dynamic analysis of a given property $\mathcal{A}$ is sound (resp. unsound) into a different program P' which is functionally equivalent to P and for which the dynamic analysis of property $\mathcal{A}$ is unsound (resp. sound).

It would also be important to extend the framework in order to take into account the feasibility of the considered coverage criterion, maybe defining some constraints that a program has to satisfy in order to guarantee the feasibility of a given coverage criterion, or by modelling and measuring situations when full coverage is not possible.

References

1. Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.
2. Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016.

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

4 Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

5 Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 643–659. USENIX Association, 2017.

6 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.*, 4(POPL):28:1–28:28, 2020.

7 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

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

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

10 Kevin Coogan, Gen Lu, and Saumya K. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 275–284. ACM, 2011.

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

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

13 P. Cousot and R. Cousot. A constructive characterization of the lattices of all retractions, preclosure, quasi-closure and closure operators on a complete lattice. *Portug. Math.*, 38(2):185–198, 1979.

14 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.

15 M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388. ACM Press, 2007. `doi:http://doi.acm.org/10.1145/1190216.1190270`.

16 Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya K. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5):25:1–25:54, 2008.

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

18 Mila Dalla Preda, Roberto Giacobazzi, and Niccoló Marastoni. Formal framework for reasoning about the precision of dynamic analysis. In *Static Analysis, 16th International Symposium, SAS 2020*, page to appear, 2020.

19 Mila Dalla Preda and Isabella Mastroeni. Characterizing a property-driven obfuscation strategy. *J. Comput. Secur.*, 26(1):31–69, 2018.

20 Bjorn De Sutter, Christian S. Collberg, Mila Dalla Preda, and Brecht Wyseur. Software protection decision support and evaluation methodologies (dagstuhl seminar 19331). *Dagstuhl Reports*, 9(8):1–25, 2019.

21 Hanne Riis Nielson Flemming Nielson and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

22 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.

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

24 Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In *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, 2015.

25 Roberto Giacobazzi, Isabella Mastroeni, and Mila Dalla Preda. Maximal incompleteness as obfuscation potency. *Formal Asp. Comput.*, 29(1):3–31, 2017.

26 Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.

27 Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 177–189, 2019.

28 Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: an obfuscation approach using probabilistic control flows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 165–185. Springer, 2016.

29 Francesco Ranzato and Francesco Tapparo. Strong preservation as completeness in abstract interpretation. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2004.

30 Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *International workshop on information hiding*, pages 270–284. Springer, 2011.

31 Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, 2016.

32 Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 94–109. IEEE Computer Society, 2009.

33 Bernhard Steffen and Gerhard J. Woeginger, editors. *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*. Springer, 2019.

34 Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE Computer Society, 2015.