Formal framework for reasoning about the precision of dynamic analysis

Mila Dalla Preda, Roberto Giacobazzi, and Niccoló Marastoni

Dipartimento di Informatica, University of Verona

Abstract. Dynamic program analysis is extremely successful both in code debugging and in malicious code attacks. Fuzzing, concolic, and monkey testing are instances of the more general problem of analysing programs by dynamically executing their code with selected inputs. While static program analysis has a beautiful and well established theoretical foundation in abstract interpretation, dynamic analysis still lacks such a foundation. In this paper, we introduce a formal model for understanding the notion of precision in dynamic program analysis. It is known that in sound-by-construction static program analysis the precision amounts to completeness. In dynamic analysis, which is inherently unsound, precision boils down to a notion of coverage of execution traces with respect to what the observer (attacker or debugger) can effectively observe about the computation. We introduce a topological characterisation of the notion of coverage relatively to a given (fixed) observation for dynamic program analysis and we show how this coverage can be changed by semantic preserving code transformations. Once again, as well as in the case of static program analysis and abstract interpretation, also for dynamic analysis we can morph the precision of the analysis by transforming the code. In this context, we validate our model on well established code obfuscation and watermarking techniques. We confirm the efficiency of existing methods for preventing control-flow-graph extraction and data exploit by dynamic analysis, including a validation of the potency of fully homomorphic data encodings in code obfuscation.

1 Introduction

Program analysis allows us to infer information on programs behaviour (semantics). It is well known from the Rice theorem that, in general, it is not possible to decide whether a given program satisfies a semantic property. For this reason analysts recur to approximation either by static or dynamic analysis. 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 on an approximation of program semantics. For this reason what we can conclude regarding the semantic property 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 [10, 11, 20]. Dynamic analysis is precise when it is sound (no false negatives), this happens when the execution traces considered by the dynamic analysis exhibit all the behaviours of the program that are relevant wrt the semantic property of interest. Code coverage is the metric typically used by dynamic analysis to evaluate its soundness, namely the amount of false negatives [1].

Program analysis has been originally developed for program verification and debugging and researchers have put a great effort in developing efficient analysis techniques and tools that reduce the number of both false positives and false negatives. In this setting, analysis precision relates to the ability of identifying bugs and vulnerabilities that may lead to unexpected behaviours, or that may be exploited by an adversary for malicious purposes.

Software protection is another interesting scenario where program analysis plays a central role but in a dual way. Indeed, in the software protection scenario program analysis is 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, the intellectual property and integrity of programs is guaranteed when the analysis is imprecise or very expensive since this complicates the attacks. In this setting, researchers have developed program transformations, called code obfuscations, with the explicit intent of complicating program analysis. In the last years many different kinds of obfuscation techniques and tools have been proposed [5]. Code obfuscation proved its efficiency in degrading the results of static program analysis while it is less efficient with respect to dynamic program analysis [28].

For example, consider a program whose control flow graph is depicted on the left of Fig. 1 where we have three blocks of sequential instructions A, B and C executed in the order specified by the arrows $\mathsf{A} \to \mathsf{B} \to \mathsf{C}$. A true opaque predicate OP^{T} is a predicate that always evaluates to *true*, but this invariant behaviour is not known to the attacker that considers as possible also the execution of the false branch [6]. In the middle of Fig. 1 we can see what happens to the control flow graph when we insert a true opaque predicate, where block D has to be considered in the static analysis of the control flow even if it is never executed at runtime. Thus, $\mathsf{A} \to OP^{\mathsf{T}} \to \mathsf{D} \to \mathsf{C}$ is a false positive path added by obfuscation to static analysis, while no imprecision is added to dynamic analysis since all executions follow the path $\mathsf{A} \to OP^{\mathsf{T}} \to \mathsf{B} \to \mathsf{C}$. On the right of

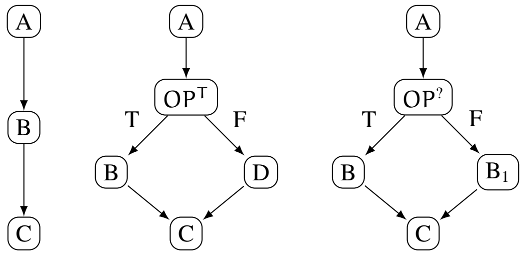


Fig. 1: Code obfuscation

Fig. 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 different but functionally equivalent (e.g. blocks B and B_1) [6]. Observe that this transformation adds confusion to dynamic analysis: a dynamic analyser has to consider more execution traces in order to observe all possible program behaviours. Indeed, if the dynamic analysis observes only traces that follow the original path $\mathsf{A} \to OP^? \to \mathsf{B} \to \mathsf{C}$ it may not be sound as it misses the traces that follow $\mathsf{A} \to OP^? \to \mathsf{B}_1 \to \mathsf{C}$ (false negative).

The abstract interpretation framework has been used to formally prove the efficiency of code obfuscation in making static analysers imprecise [13]. Indeed, code obfuscation hampers static analysis by exploiting its conservative nature, namely by increasing its imprecision (false positives) while preserving the program intended behaviour. It has been observed that adding false positives to the analysis can be formalised in terms of incompleteness in the analysis of the transformed program [13, 14, 18, 21]. Observe that, in general, the imprecision added by these obfuscating transformations in order to confuse a static analyzer is not able to confuse a dynamic attacker that looks at the real program execution and thus cannot be deceived by false positives. Indeed, dynamic analysis observes only paths that are actually executed. For this reason common deobfuscation approaches often recur to dynamic analysis to understand obfuscated code [3, 7, 31, 38].

It is clear that to hamper dynamic analysis we need to develop obfuscation techniques that exploit the Achilles heel of dynamic analysis and that increases the number of false negatives. In the literature, there are defense techniques that focus on hampering dynamic analysis [2, 25–27]. We would like to provide a formal framework where it is possible to prove and discuss the efficiency of these techniques in complicating dynamic analysis in terms of the imprecision (false negatives) that they introduce in the analysis. This will allow us to better understand the potential and limits of code obfuscation against dynamic program analysis. We start by providing a formalisation of dynamic analysis and software protection techniques in terms of program semantics and equivalence reactions over semantic domains, and we characterise when a program transformation hampers a dynamic analysis in terms of topological features.

The contribution of this work are: (1) formal specification for dynamic analysis/attacks based on program semantics and equivalence relations; (2) formal definition of software-based protection transformations against dynamic attacks that induce imprecision in dynamic analysis (false negatives); (3) validation of the model on some known software-based defense strategies.

2 Preliminaries

Basic lattice and fix-point theory: Given two sets S and T, we denote with $\wp(\mathsf{S})$ the powerset of S, with $\mathsf{S} \times \mathsf{T}$ the Cartesian product of S and T, with $\mathsf{S} \subset \mathsf{T}$ strict inclusion, with $\mathsf{S} \subseteq \mathsf{T}$ inclusion, with $\mathsf{S} \subseteq_{\mathsf{F}} \mathsf{T}$ the fact that S is a finite set. $\langle \mathsf{C}, \leqslant, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice on the set C, with ordering $\leqslant$, least upper bound (*lub*) $\vee$, greatest lower bound (*glb*) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. Let C and D be complete lattices. Then, $\mathsf{C} \xrightarrow{\mathrm{m}} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\mathrm{c}} \mathsf{D}$ 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 $\mathit{lfp}(f) = \bigvee_{i \in \mathbb{N}} f^i(\bot_C)$, where, for any $i \in \mathbb{N}$ and $x \in C$, the i-th power of f in x is inductively defined as follows: $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$.

Given a relation $\mathcal{R} \subseteq C \times D$ between two sets C and D, and two elements $x \in C$ and $y \in D$, then $(x, y) \in \mathcal{R}$ denotes that the pair (x, y) belongs to the relation $\mathcal{R}$. A binary relation $\mathcal{R}$ on a set C, namely $\mathcal{R} \subseteq C \times C$, 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 R. Let $\mathit{eq}(C)$ be the set of equivalence relations over the set C. The equivalence classes of an equivalence relation $\mathcal{R} \in \mathit{eq}(C)$ form 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 set of equivalence classes of relation $\mathcal{R}$ is called the quotient set of C and it is denoted by $C/_\mathcal{R}$. A partition $C/_{\mathcal{R}_1}$ is a refinement of a partition $C/_{\mathcal{R}_2}$, namely $\mathcal{R}_1$ if finer than $\mathcal{R}_2$ or $\mathcal{R}_2$ is coarser than $\mathcal{R}_1$, if every equivalence class in $C/_{\mathcal{R}_1}$ is a subset of some equivalence class in $C/_{\mathcal{R}_2}$. We denote with $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ the fact that the equivalence relation $\mathcal{R}_1$ is finer than the equivalence relation $\mathcal{R}_2$. Given a subset $S \subseteq C$ we denote with $\mathcal{R}(S)$ the set of equivalence classes of the elements of S, namely $\mathcal{R}(S) = \{[x]_\mathcal{R} \mid x \in S\}$, and with $S/_\mathcal{R}$ the partition of the subset S induced by the equivalence relation $\mathcal{R}$, namely $S/_\mathcal{R} = \{[x]_\mathcal{R} \cap S \mid x \in S\}$.

Program semantics: Let *Prog* be a set of programs ranged over by P. Let $v \in \mathbb{I}$ denote a possible input and let $\mathbb{I}^*$ denote the set of input sequences ranged over by $\mathcal{I}$, let *PP* denote the set of program points ranged over by *pp*, let *Com* denote the set of program statements ranged over by C and let *Mem* denote the set of memory maps that associates values to variables ranged over by $m : \mathit{Var} \to \mathit{Values}$. $\Sigma = \mathbb{I}^* \times \mathit{PP} \times \mathit{Com} \times \mathit{Mem}$ is the set of program states. Thus, a program state $s \in \Sigma$ is a tuple $s = \langle \mathcal{I}, \mathit{pp}, C, m \rangle$ where $\mathcal{I}$ denotes the sequence of inputs that still needs to be consumed to terminate the execution, *pp* denotes the program point of the next instruction C that has to be executed, and m is the current memory. We denote with $C_1; C_2$ the sequential composition of statements and we refer to *skip* as the identity statement whose execution has no effects on memory. Given a program P we denote with $\mathbb{I}_P \subseteq \mathbb{I}^*$ the set of the initial input sequences for the execution of program P, and with $\mathit{Init}_P = \{s \in \Sigma \mid s = \langle \mathcal{I}, \mathit{pp}, C, m \rangle, \mathcal{I} \in \mathbb{I}_P\}$ the set of its initial states. We use Σ^* to denote the set of all finite and infinite sequences or traces of states, where $\epsilon \in \Sigma^*$ is the empty sequence, $|\sigma|$ the length of sequence $\sigma \in \Sigma^*$. $\Sigma^+ \subset \Sigma^*$ denotes the set of finite sequences of elements of Σ. We denote the concatenation of sequences $\sigma, \nu \in \Sigma^*$ as $\sigma\nu$. Given $\sigma, \nu \in \Sigma^*$, $\nu \preceq \sigma$ means that ν is a subsequence of σ, namely that there exists $\sigma_1, \sigma_2 \in \Sigma^*$ such that $\sigma = \sigma_1 \nu \sigma_2$. Given $s \in \Sigma$ we write $s \in \sigma$ when s is an element occurring in sequence σ, and we denote with $\sigma_0 \in \Sigma$ the first element of sequence σ and with σ_f the final element of the finite sequence $\sigma \in \Sigma^+$. Let $R \subseteq \Sigma \times \Sigma$ denote the transition relation between program states, thus $(s, s') \in R$ means that state s' can be obtained from state s in one computational step. The *(finite) trace semantics* of a program P is defined, as usual, as the least fix-point computation of function $\mathcal{F}_P : \wp(\Sigma^*) \to \wp(\Sigma^*)$ [9]:

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

The trace semantics of P is $[\![P]\!] = \mathit{lfp}(\mathcal{F}_P) = \bigcup_{i \in \mathbb{N}} \mathcal{F}_P^i(\bot_C)$. $\mathit{Den}[\![P]\!]$ 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. Therefore we have $\mathit{Den}[\![P]\!] = \{\sigma \in \Sigma^+ \mid \exists \eta \in [\![P]\!] : \eta_0 = \sigma_0, \eta_f = \sigma_f\}$.

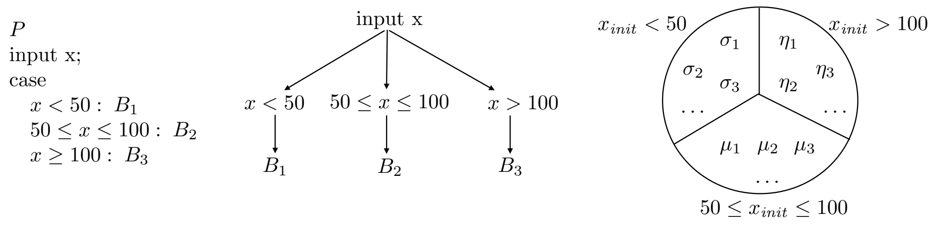
3 Topological characterisation of the precision of dynamic analysis

We start our investigation by considering dynamic analysis that observes features of single execution traces, as for example: 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, etc. The extension of the framework to properties of sets of traces (hyper-properties) and relational properties among traces is left as future work.

The simplest way to model properties of single traces is in terms of equivalence relations over program traces. Indeed, an equivalence relation $\mathcal{R} \in \mathit{eq}(\Sigma^*)$ groups together all those execution traces that are equivalent wrt the property used to establish the equivalence for $\mathcal{R}$. In this setting, each equivalence class $[\sigma]_{\mathcal{R}} \subseteq \Sigma^*$ represents the set of execution traces that are equivalent to σ wrt $\mathcal{R}$, namely all those execution traces that $\mathcal{R}$ is not able to distinguish from σ. In general, given a program $P \in \mathit{Prog}$ and an equivalence relation $\mathcal{R} \in \mathit{eq}(\Sigma^*)$ it may not be possible to precisely observe property $\mathcal{R}$ of program semantics, namely the set $\mathcal{R}([\![P]\!]) = \{[\sigma]_{\mathcal{R}} \mid \sigma \in [\![P]\!]\}$ may not be precisely observable. This means that it may not be possible to decide whether $\mathcal{R}([\![P]\!]) \subseteq \Pi$, for some $\Pi \in \wp(\Sigma^*/_{\mathcal{R}})$, a set of equivalence classes representing a possible feature of program execution that can be expressed in terms of $\mathcal{R}$. In order to verify these features, analysts resort to approximation either by static or dynamic analysis.

Example 1. Consider function $\iota : \Sigma \to \mathbb{I}$ that observes the first input value $\nu \in \mathbb{I}$ of a program state, namely $\iota(\langle \nu\mathcal{I}, pp, C, m\rangle) \overset{\text{def}}{=} \nu$. We can define the equivalence relation $\mathcal{R}_\iota$ as $(\sigma, \nu) \in \mathcal{R}_\iota$ iff $\iota(\sigma_0) = \iota(\nu_0)$, grouping together traces with the same starting input values. Based on $\mathcal{R}_\iota$ we can define features of program behaviour as for example $\Pi_1, \Pi_2 \in \wp(\Sigma^*/_{\mathcal{R}_\iota})$ where $\Pi_1 = \{[\sigma]_{\mathcal{R}_\iota} \mid \iota(\sigma) \geqslant 0\}$ observes the equivalence classes of traces whose first input value is positive, and $\Pi_2 = \{[\sigma]_{\mathcal{R}_\iota} \mid \iota(\sigma) \in [l, u]\}$ observes the equivalence classes of traces whose first input value is in the interval $[l, u]$.

We can think about relation $\mathcal{R}$ as the granularity at which the analysis observes program executions. Given $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ we have that $\mathcal{R}_1$ describes an analysis that is more precise than $\mathcal{R}_2$ in distinguishing program traces, while $\mathcal{R}_2$ describes an analysis that groups together more traces than $\mathcal{R}_1$. The equivalence classes can then be combined to describe properties of programs at different levels of abstraction.



```
P
input x;
case
  x < 50 : B1
  50 ≤ x ≤ 100 : B2
  x ≥ 100 : B3
```

input x
$x < 50$
$50 \leq x \leq 100$
$x > 100$
B_1
B_2
B_3
$x_{init} < 50$
$x_{init} > 100$
σ_1 σ_2 σ_3 ...
η_1 η_2 η_3 ...
μ_1 μ_2 μ_3 ...
$50 \leq x_{init} \leq 100$

Fig. 2: Dynamic analysis and soundness

In the literature there exists a formal investigation of the effects of code obfuscation to the precision of static analysis [13, 14, 18, 21]. This has lead to a better understanding of the potential and limits of obfuscation, and it has been useful in the design of obfuscation techniques that target specific program properties [14, 18, 19].

In the following we apply a similar approach to dynamic analysis. To this end we formalise the absence of false negatives, namely the precision of dynamic analysis, in terms of topological properties of program trace semantics and of the equivalence relation $\mathcal{R}$ modelling the property to be observed. False negatives happen when the set of traces considered by dynamic analysis misses some traces that would modify the equivalence classes observed by property $\mathcal{R}$. We show how to transform a program in order to hinder the dynamic analysis of a property $\mathcal{R}$, namely in order to make the dynamic analysis of the transformed program not sound.

3.1 Modelling dynamic program analysis

Dynamic analysis observes a finite subset of finite execution traces of a program and from this partial observation tries to drive conclusions on the whole program behaviour.

Definition 1 (Dynamic Execution). *The execution traces of program* P *with initial states in* $\mathsf{T_P} \subseteq_{\mathsf{F}} \mathit{Init}_{\mathsf{P}}$ *and with time limits* $\mathsf{t} \in \mathbb{N}$*, are defined as:*

$$\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \overset{def}{=} \left\{ \sigma \in [\![\mathsf{P}]\!] \;\middle|\; |\sigma| \leqslant \mathsf{t}, \sigma = s_0\sigma', s_0 \in \mathsf{T_P} \right\}$$

Note that $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ is a finite set and that each trace in $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ is finite (it has at most t states). This correctly implies that: $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \subseteq_{\mathsf{F}} [\![\mathsf{P}]\!]$. The goal of dynamic analysis is to derive knowledge of a semantic property of a program by observing a finite subset $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ of its execution traces. Dynamic analysis is therefore specified as the set of observed execution traces $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ and of an equivalence relation on traces $\mathcal{R} \in \mathit{eq}(\Sigma^*)$.

Definition 2 (Dynamic Analysis). *A dynamic analysis of property* $\mathcal{R} \in \mathit{eq}(\Sigma^*)$ *of program* $\mathsf{P} \in \mathit{Prog}$*, is defined as a pair* $\langle \mathcal{R}, \mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \rangle$.

Let us consider program P on the left of Fig. 2 where the block of code to execute depends on the input value of x. Consider a property of traces $\bar{\mathcal{R}} \in \mathit{eq}(\Sigma^*)$ that observes which block B_1, B_2 or B_3 of program P is executed. On the right of Fig. 2 we represent the partition of the traces of program P induced by property $\bar{\mathcal{R}}$ where x_{Init} denotes the input value of variable x.

Dynamic analysis $\langle \mathcal{R}, \mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \rangle$ can precisely observe property $\mathcal{R}$ of the semantics of P (no false negatives) when $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ contains at least one trace for each one of the equivalence classes of the traces of $[\![\mathsf{P}]\!]$.

Definition 3 (Soundness). *Given* $\mathsf{P} \in \mathit{Prog}$ *and* $\mathcal{R} \in \mathit{eq}(\Sigma^*)$ *a dynamic analysis* $\langle \mathcal{R}, \mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \rangle$ *is sound if* $\forall \mathsf{x} \in [\![\mathsf{P}]\!] : [\mathsf{x}]_{\mathcal{R}} \in \mathcal{R}(\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}))$.

When a dynamic analysis $\langle \mathcal{R}, \mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \rangle$ is sound we have no false negatives, namely $\forall \mathsf{y} \in [\![\mathsf{P}]\!] : [\mathsf{y}]_{\mathcal{R}} \in \mathcal{R}(\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}))$. When this happens, all the behaviours of program P that relation $\mathcal{R}$ is able to distinguish are taken into account by the partial observation of program behaviour $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$. In the example in Fig. 2 we have that a dynamic analysis $\langle \bar{\mathcal{R}}, \mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \rangle$ is sound if $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ contains at least one execution trace for each one of the three equivalence classes depicted on the right of Fig. 2.

Definition 4 (Covers). *Given* $\mathsf{P} \in \mathit{Prog}$, *and* $\mathcal{R} \in \mathit{eq}(\Sigma^*)$, *we say that* $\mathsf{S} \subseteq [\![\mathsf{P}]\!]$ *covers* P *wrt* $\mathcal{R}$ *when:* $\mathcal{R}(\mathsf{S}) = \mathcal{R}([\![\mathsf{P}]\!])$.

It is clear that when S covers P wrt $\mathcal{R}$ we have that the partial observation S of the behaviours of P is sound wrt $\mathcal{R}$, since it allows us to observe all the equivalence classes of $\mathcal{R}$ that we would observe by having access to all the traces in $[\![\mathsf{P}]\!]$ (no false negatives). Thus, in the example in Fig. 2 we have that the set of traces $\{\sigma_1, \eta_1\}$ does not cover P wrt $\bar{\mathcal{R}}$, while the set of traces $\{\sigma_1, \eta_1, \eta_2, \mu_2\}$ does. The following theorem comes straight from the definitions.

Theorem 1. *Given* $\mathsf{P} \in \mathit{Prog}$ *and* $\mathcal{R} \in \mathit{eq}(\Sigma^*)$, *if* $\mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ *covers* P *wrt* $\mathcal{R}$ *then the dynamic analysis* $\langle \mathcal{R}, \mathit{Exe}(\mathsf{P}, \mathsf{T_P}, \mathsf{t}) \rangle$ *is sound (no false negatives).*

The goal of dynamic analysis of a property $\mathcal{R}$ on a program P, is to identify the set $\mathsf{T_P}$ of inputs, and the length t that induce a partial observation of program semantics that makes the analysis sound (no false negatives) wrt $\mathcal{R}$. Thus, a possible way to hamper dynamic analysis is to transform programs in order to increase the number of traces that it is necessary to observe to ensure soundness. Indeed, by tying the precision of dynamic analysis to the observation of a wider set of traces (worst case being the observation of all possible traces) we are limiting the advantages of using dynamic analysis.

In order to formalise this idea, in the following we provide a characterisation of the set of traces that are needed to guarantee the soundness of the dynamic analysis of a program P wrt a semantic property $\mathcal{R}$. We use this characterisation to formalise what it means for a software-based defense transformation to harm dynamic analysis. We validate our model by showing how it naturally relates to the notion of code coverage of dynamic analysis, and by showing how existing techniques for hindering dynamic analysis fit in our framework.

3.2 Harming Dynamic Analysis

Given an equivalence relation $\mathcal{R} \in \mathit{eq}(\Sigma^*)$ concerning what we can observe and a set of equivalence classes $\mathsf{X} \in \wp(\Sigma^*/_{\mathcal{R}})$ we would like to characterise the minimal sets of traces that the relation $\mathcal{R}$ maps to X.

Definition 5 (Core). *Consider* $\mathcal{R} \in eq(\Sigma^*)$ *and* $X \in \wp(\Sigma^*/_\mathcal{R})$*:*

$$Core(X,\mathcal{R}) \stackrel{def}{=} \left\{ T = \{\sigma \in \Sigma^* \mid [\sigma]_\mathcal{R} \in X\} \,\middle|\, \begin{array}{l} \forall \sigma_1, \sigma_2 \in T, \sigma_1 \neq \sigma_2 \Rightarrow [\sigma_1]_\mathcal{R} \neq [\sigma_2]_\mathcal{R} \\ \forall [\nu]_\mathcal{R} \in X : \exists \sigma \in T : [\sigma]_\mathcal{R} = [\nu]_\mathcal{R} \end{array} \right\}$$

Theorem 2. *Consider* $\mathcal{R} \in eq(\Sigma^*)$ *and* $X \in \wp(\Sigma^*/_\mathcal{R})$*:*

1. *Given* $T \in Core(X,\mathcal{R})$ *we have that:* $\mathcal{R}(T) = X$
2. $\forall S \in \wp(\Sigma^*)$*: If* $\mathcal{R}(S) = X$ *then* $\exists T \in Core(X,\mathcal{R}) : T \subseteq S$

This means that $Core(\mathcal{R}(\llbracket P \rrbracket),\mathcal{R})$ characterises the minimal sets of execution traces that provide a sound dynamic analysis of property $\mathcal{R}$ for program P. In the example in Fig. 2 we have that $Core(\llbracket P \rrbracket, \bar{\mathcal{R}})$ identifies those sets of trace that have exactly three traces: one trace with $x_{init} < 50$, one trace with $50 \leqslant x_{init} \leqslant 100$ and one trace with $x_{init} > 100$.

Corollary 1. *Given* $P \in Prog$ *and* $\mathcal{R} \in eq(\Sigma^*)$ *we have that:*

– $\forall T \in Core(\mathcal{R}(\llbracket P \rrbracket),\mathcal{R})$ *we have that* T *covers* $\llbracket P \rrbracket$ *wrt* $\mathcal{R}$.
– *Given* $T_P \subseteq_F Init_P$ *and* $t \in \mathbb{N}$ *the dynamic analysis* $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$ *is sound iff* $\exists T \in Core(\mathcal{R}(\llbracket P \rrbracket),\mathcal{R})$ *such that* $T \subseteq Exe(P, T_P, t)$.
– *For every semantic feature* $\Pi \in \wp(\Sigma^*/_\mathcal{R})$ *expressed in terms of equivalence classes of* $\mathcal{R}$*, we have that if* $Exe(P, T_P, t)$ *covers* $\llbracket P \rrbracket$ *wrt* $\mathcal{R}$ *then we can precisely evaluate* $\llbracket P \rrbracket \subseteq \Pi$ *by evaluating* $Exe(P, T_P, t) \subseteq \Pi$.

Thus, a dynamic analysis $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$ is sound if $Exe(P, T_P, t)$ observes at least one execution trace for each one of the equivalence classes of the traces in $\llbracket P \rrbracket$ for the relation $\mathcal{R}$. In the worst case we have a different equivalence class for every execution trace of P. When this happens, a sound dynamic analysis of property $\mathcal{R}$ on program P has to observe all possible execution traces, which is unfeasible in the general case. Thus, if we want to protect a program from a dynamic analysis that is interested in the property $\mathcal{R}$, we have to diversify property $\mathcal{R}$ as much as possible among the execution traces of the program.

This allows us to define when a program transformation is *potent* wrt a dynamic analysis, namely when a program transformation forces a dynamic analysis to observe a wider set of traces in order to be sound. See [5] for the general notion of potency of a program transformation, i.e., a program transformation that foils a given attack (in our case a dynamic analysis).

Definition 6 (Potency). *A program transformation* $\mathcal{T} : Prog \to Prog$ *that preserves the denotational semantics of programs is potent for a program* $P \in Prog$ *wrt an observation* $\mathcal{R} \in eq(\Sigma^*)$ *if the following two conditions hold:*

1. $\forall \sigma_1, \sigma_2 \in \llbracket \mathcal{T}(P) \rrbracket : [\sigma_1]_\mathcal{R} = [\sigma_2]_\mathcal{R}$ *we have that* $\forall \nu_1, \nu_2 \in \llbracket P \rrbracket : Den(\nu_1) = Den(\sigma_1) \wedge Den(\nu_2) = Den(\sigma_2)$ *then* $[\nu_1]_\mathcal{R} = [\nu_2]_\mathcal{R}$
2. $\exists \nu_1, \nu_2 \in \llbracket P \rrbracket : [\nu_1]_\mathcal{R} = [\nu_2]_\mathcal{R}$ *for which* $\exists \sigma_1, \sigma_2 \in \llbracket \mathcal{T}(P) \rrbracket : Den(\nu_1) = Den(\sigma_1) \wedge Den(\nu_2) = Den(\sigma_2)$ *such that* $[\sigma_1]_\mathcal{R} \neq [\sigma_2]_\mathcal{R}$

Fig. 3 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

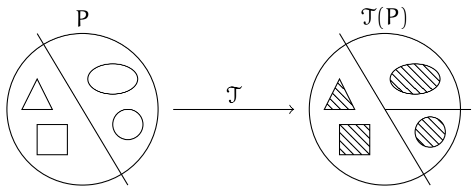


Fig. 3: Transformation Potency

$\mathcal{R}$, while on the right we have the traces of the transformed program $\mathcal{T}(P)$ partitioned according to $\mathcal{R}$. Traces that are denotationally equivalent have the same shape (triangle, square, circle, oval), but are filled differently since they are in general different traces. The first condition means that the traces of $\mathcal{T}(P)$ that property $\mathcal{R}$ maps to the same equivalence class (triangle and square), are denotationally equivalent to traces of P that property $\mathcal{R}$ maps to the same equivalence class. This means that what is grouped together by $\mathcal{R}$ on $[\![\mathcal{T}(P)]\!]$ was grouped together by $\mathcal{R}$ on $[\![P]\!]$, modulo the denotational equivalence of traces. The second condition requires that there are traces of P (circle and oval) that property $\mathcal{R}$ maps to the same equivalence class and whose denotationally equivalent traces in $\mathcal{T}(P)$ are mapped by $\mathcal{R}$ to different equivalence classes. This means that a defense technique against dynamic analysis wrt a property $\mathcal{R}$ is successful when it transforms a program into a functionally equivalent one for which property $\mathcal{R}$ 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 $\mathcal{R}$ varies at every execution trace.

Example 2. Consider the following programs P and Q that compute the sum of natural numbers from $x \geqslant 0$ 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 $\langle \mathcal{I}, pp, C, [val_x, val_{sum}] \rangle$, where 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:

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

This allows us to define the equivalence relation $\mathcal{R}_{Max} \in eq(\Sigma^*)$ that observes traces wrt the maximal value assumed by x at $\bullet$, as $(\sigma, \sigma') \in \mathcal{R}_{Max}$ iff $Max(\sigma) = Max(\sigma')$. The equivalence classes of $\mathcal{R}_{Max}$ are the sets of traces with the same maximal value assumed by x at $\bullet$. We can observe that all the execution traces of P belong to the same equivalence class of $\mathcal{R}_{Max}$. In this case, a dynamic analysis $\langle \mathcal{R}_{Max}, Exe(P, T_P, t) \rangle$ is sound if $Exe(P, T_P, t)$ 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. Q is equivalent to P, i.e., $Den[\![P]\!] = Den[\![Q]\!]$, but 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 $\mathcal{R}_{Max}$ 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 $\mathcal{R}_{Max}$ depending on the value assumed by n. Thus, $\mathcal{R}([\![Q]\!])$ 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 $\langle \mathcal{R}_{Max}, Exe(Q, T_Q, t) \rangle$ is sound if $Exe(Q, T_Q, t)$ contains at least one execution trace for each of the possible values of n generated during execution.

4 Model Validation

In this section we show how the proposed framework can be used to model existing code obfuscation techniques. In particular we model the way these transformations deceive dynamic analysis of control flow and data flow properties of programs. We also show how the measures of code coverage used by dynamic analysis tools can be naturally interpreted in the proposed framework.

4.1 Control Flow Analysis

Dynamic Extraction of the Control Flow Graph. The control flow graph CFG of a program P is a graph $CFG_P = (V, E)$ where each node $v \in V$ is a pair (pp, C) denoting a statement C at program point pp in P, and $E \subseteq V \times V$ is the set of edges such that $(v_1, v_2) \in E$ means that the statement in v_2 could be executed after the statement in v_1 when running P. Thus, we define the domain of nodes as $Nodes \overset{\text{def}}{=} PP \times Com$, and the domain of edges as $Edges \overset{\text{def}}{=} Nodes \times Nodes$. It is possible to dynamically construct the CFG of a program by observing the commands that are executed and the edges that are traversed when the program runs. Let us define $\eta : \Sigma \rightarrow Nodes$ that observes the command to be executed together with its program point, namely

$\eta(s) = \eta(\langle \mathcal{I}, pp, C, m\rangle) \stackrel{\text{def}}{=} (pp, C)$. By extending this function on traces we obtain function *path* : $\Sigma^* \to$ *Nodes* $\times$ *Edges* that extracts the path of the CFG corresponding to the considered execution trace, abstracting from the number of times that an edge is traversed or a node is computed:

$$path(\sigma) \stackrel{\text{def}}{=} (\{\eta(s) \mid s \in \sigma\}, \{(\eta(s), \eta(s')) \mid ss' \preceq \sigma\})$$

where $s \in \sigma$ means that s is a state that appears in trace σ and $ss' \preceq \sigma$ means that s and s' are successive states in σ. This allows us to define the equivalence relation $\mathcal{R}_{CFG} \in eq(\Sigma^*)$ that observes traces up to the path that they define, as $(\sigma, \sigma') \in \mathcal{R}_{CFG}$ iff $path(\sigma) = path(\sigma')$. Indeed, $\mathcal{R}_{CFG}$ groups together those traces that execute the same set of nodes and traverse the same set of edges, abstracting from the number of times that nodes are executed and edges are traversed.

The CFG of a program P can be defined as the union of the paths of its execution traces, namely $CFG_P = \bigsqcup\{path(\sigma) \mid \sigma \in [\![P]\!]\}$, where the union of graphs is defined as $(V_1, E_1) \sqcup (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$. The dynamic extraction of the CFG of a program P from the observation of a set $X \subseteq_F [\![P]\!]$ of execution traces, is given by $\bigsqcup\{path(\sigma) \mid \sigma \in X\}$. In the general case we have $\bigsqcup\{path(\sigma) \mid \sigma \in X\} \subseteq CFG_P$.

Preventing Dynamic CFG Extraction. Control code obfuscations are program transformations that modify the program's control flow in order to make it difficult for an adversary to analyse the flow of control of programs [5]. According to Section 3.2, a program transformation $\mathcal{T}$: *Prog* $\to$ *Prog* is a potent defence against the dynamic extraction of the CFG of a program P when $\mathcal{T}$ diversifies the paths taken by the execution traces of $\mathcal{T}(P)$ wrt the paths taken by the traces of P. In the following, we show how two known defence techniques for preventing dynamic analysis actually work by diversifying program traces with respect to property $\mathcal{R}_{CFG}$.

Range Dividers: Range Divider (*RD*) is a transformation designed to prevent dynamic symbolic execution and it is an efficient protection against the dynamic extraction of the CFG [2]. *RD* relies on the existence of n program transformations $\mathcal{T}_i$: *Prog* $\to$ *Prog* with $i \in [1, n]$ that:

1. Preserve the denotational semantics of programs:
$$\forall P \in Prog, i \in [1, n] : Den[\![P]\!] = Den[\![\mathcal{T}_i(P)]\!]$$
2. Modify the paths of the CFG of programs in different ways:
$$\forall P \in Prog, \forall i, j \in [1, n]: \ \mathcal{R}_{CFG}([\![\mathcal{T}_i(P)]\!]) = \mathcal{R}_{CFG}([\![\mathcal{T}_j(P)]\!]) \Rightarrow i = j.$$

Given a program P, the *RD* transformation works by inserting a `switch` control statement with n cases and whose condition depends on program inputs. Every case of the `switch` contains a semantically equivalent version $\mathcal{T}_i(P)$ of P that is specialised wrt the input values. Thus, depending on the input values we would execute one of the diversified programs $\mathcal{T}_1(P), \dots, \mathcal{T}_n(P)$. Since for each variant $\mathcal{T}_i(P)$ with $i \in [1, n]$ the set of execution traces are mapped by $\mathcal{R}_{CFG}$ into different equivalent classes, we have that property $\mathcal{R}_{CFG}$ has been diversified among the traces of $RD(P)$. Thus, the transformation *RD* is potent wrt $\mathcal{R}_{CFG}$ and harms the dynamic extraction of the CFG.

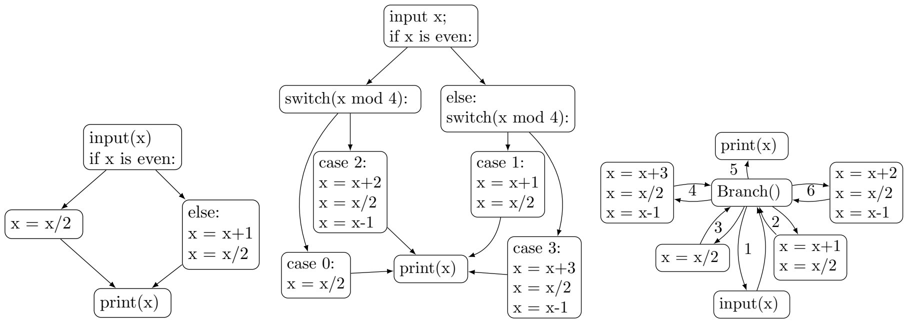


Fig. 4: CFG of P, $RD(\mathsf{P})$ and $GD(\mathsf{P})$

A simple example is provided in Fig. 4 where on the left we have the CFG of the original program P. P verifies the parity of the input value and then computes the integer division. The second graph in Fig. 4 represents the CFG of program P transformed by *RD*. The CFG of program $RD(\mathsf{P})$ has four different paths depending on the value of the input variable x. Each one of these paths is functionally equivalent to the corresponding path in P (`case 0` and `case 2` are equivalent to the path taken when x is even, while `case 1` and `case 3` are equivalent to the path taken when x is odd). We can easily observe that in this case the paths of $RD(\mathsf{P})$ have been diversified wrt the paths of P. Indeed, a dynamic analysis has to observe two execution traces to precisely build the CFG for P, while four traces are need to precisely build the CFG of $RD(\mathsf{P})$.

Gadget Diversification: In [27] the authors propose a program transformation, denoted $GD : Prog \to Prog$ that hinders the dynamic CFG analysis. *GD* starts by identifying a sequence Q_{seq} of sequential command (no branches) in program P. Next, *GD* assumes to have access to a set of diversifying transformations $\mathcal{T}_i : Prog \to Prog$ with $i \in [1, n]$ that diversify command sequences while preserving their functionality. These transformations are then applied to portions of Q_{seq} in order to generate a wide set $\mathsf{S}_{seq} = \{\mathsf{Q}_1..\mathsf{Q}_m\}$ of command sequences where each $\mathsf{Q}_j \in \mathsf{S}_{seq}$ is functionally equivalent to Q_{seq}, while every pair $\mathsf{Q}_j, \mathsf{Q}_l \in \mathsf{S}_{seq}$ are such that $\mathcal{R}_{CFG}(\llbracket \mathsf{Q}_j \rrbracket) \neq \mathcal{R}_{CFG}(\llbracket \mathsf{Q}_l \rrbracket)$. This means that each execution trace generated by the run of a sequence in S_{seq} belongs to a different equivalence class wrt relation $\mathcal{R}_{CFG}$, while being denotationally equivalent by definition.

Transformation *GD* proceeds by adding a `branching function` to the original program P that, depending on the input values, deviates the control flow to one of the sequences of commands in S_{seq}. Thus, depending on the input values, *GD* diversifies the path that is executed. This makes the transformation *GD* potent wrt $\mathcal{R}_{CFG}$ according to the proposed framework.

A simple example of *GD* can be observed in the third graph of Fig. 4, where the original program is transformed to reveal a peculiar CFG structure. The `branch` function is here symbolized as the central block from which all other blocks are called and to which all other blocks return (except for `print(x)` which represents the end of the program). The `branch` function will only allow the following sequences of edges:

$$\mathsf{odd}(\mathsf{x}) \to \left\{ \begin{matrix} 1 \to 2 \to 5 \\ 1 \to 4 \to 5 \end{matrix} \right\} \qquad \mathsf{even}(\mathsf{x}) \to \left\{ \begin{matrix} 1 \to 3 \to 5 \\ 1 \to 6 \to 5 \end{matrix} \right\}$$

We can easily observe that the paths of $GD(\mathsf{P})$ have been diversified wrt the paths of P and while the dynamic construction of the CFG for P requires to observe two execution traces, we need to observe 4 execution traces to precisely build the CFG of $GD(\mathsf{P})$.

4.2 Code Coverage

Most dynamic algorithms use code coverage to measure the potential soundness of the analysis [1]. Intuitively, given a program P and a partial observation $Exe(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ of its execution traces, code coverage wants to measure the amount of program behaviour considered by $Exe(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$ wrt the set of all possible behaviours $[\![\mathsf{P}]\!]$. In the following we describe some known code coverage measures.

Statement coverage considers the statements of the program that have been executed by the traces in $Exe(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$. This is a function $st : \Sigma^* \to Nodes$ that collects commands annotated with their program point, that are executed along a considered trace: $st(\sigma) \overset{\text{def}}{=} \{\eta(s) \mid s \in \sigma\}$. This allows us to define the equivalence relation $\mathcal{R}_{st} \in eq(\Sigma^*)$ that groups together traces that execute the same set of statements.

Count-Statement coverage considers how many times each statement of the program has been executed by the traces in $Exe(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$. Thus, it can be formalised in terms of an equivalence relation $\mathcal{R}^+_{st} \in eq(\Sigma^*)$ that groups together traces that execute the same set of statements the same amount of times. It is clear that relation $\mathcal{R}^+_{st}$ is finer than relation $\mathcal{R}_{st}$, namely $\mathcal{R}^+_{st} \sqsubseteq \mathcal{R}_{st}$.

Path coverage observes the nodes executed and edges traversed by the traces in $Exe(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$. This precisely corresponds to the observation of property $\mathcal{R}_{CFG} \in eq(\Sigma^*)$ defined above, where the paths of the CFG are observed by abstracting form the number of times that edges are traversed. It is clear that relation $\mathcal{R}_{CFG}$ is finer than relation $\mathcal{R}_{st}$, namely $\mathcal{R}_{CFG} \sqsubseteq \mathcal{R}_{st}$.

Count-Path coverage considers the different paths in $Exe(\mathsf{P}, \mathsf{T_P}, \mathsf{t})$, where the number of times that edges are traversed in a trace is taken into account. This can be formalised in terms of an equivalence relation $\mathcal{R}^+_{CFG} \in eq(\Sigma^*)$ that groups together traces that execute and traverse the same nodes and edges the same number of times. It is clear that relation $\mathcal{R}^+_{CFG}$ is finer than relation $\mathcal{R}_{CFG}$, namely $\mathcal{R}^+_{CFG} \sqsubseteq \mathcal{R}_{CFG}$.

Trace coverage considers the traces of commands that have been executed abstracting from the memory map. In this case we can define the code coverage in terms of function $trace : \Sigma^* \to Com \times PP$ defined as $trace(\epsilon) \overset{\text{def}}{=} \epsilon$ and $trace(s\sigma) \overset{\text{def}}{=} \eta(s)trace(\sigma)$. The equivalence relation $\mathcal{R}_{trace} \in eq(\Sigma^*)$ is such that $(\sigma, \sigma') \in \mathcal{R}_{trace}$ if $trace(\sigma) = trace(\sigma')$. This equivalence relation is finer than $\mathcal{R}^+_{CFG}$ since it keeps track of the order of execution of the edges.

In order to avoid false negatives, dynamic algorithms automatically look for inputs whose execution traces have to exhibit new behaviours with respect to the code coverage metric used (e.g., they have to execute new statements or execute them a different number of times, traverse new edges or change the number of times edges are traversed, or execute nodes in a different order). This can be naturally formalised in our framework. Given a set $Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t})$ of observed traces, an automatically generated input increases the code coverage measured as $\mathcal{R}_{st}$ (or $\mathcal{R}_{st}^+, \mathcal{R}_{CFG}, \mathcal{R}_{CFG}^+, \mathcal{R}_{trace}$) if the execution trace σ generated by the input is mapped in a new equivalence class of $\mathcal{R}_{st}$ (or $\mathcal{R}_{st}^+, \mathcal{R}_{CFG}, \mathcal{R}_{CFG}^+, \mathcal{R}_{trace}$), namely in an equivalence class that was not observed by traces in $Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t})$, namely if $[\sigma]_{\mathcal{R}_{st}} \notin \mathcal{R}_{st}(Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t}))$ (analogously for $\mathcal{R}_{st}^+$, $\mathcal{R}_{CFG}$, $\mathcal{R}_{CFG}^+$, $\mathcal{R}_{trace}$). We have seen above that some of the common measures for code coverage can be expressed in terms of semantic program properties with different degrees of precision $id \sqsubseteq \mathcal{R}_{traces} \sqsubseteq \mathcal{R}_{CFG}^+ \sqsubseteq \mathcal{R}_{CFG} \sqsubseteq \mathcal{R}_{st}$. This means, for example, that automatically generated inputs could add coverage for $\mathcal{R}_{CFG}^+$ but not for $\mathcal{R}_{st}$. Indeed, a new input generates a new behaviour depending on the metric used for code coverage.

Fuzzing and dynamic symbolic execution are typical techniques used by dynamic analysis to automatically generate inputs in order to extend code coverage. The metrics that fuzzing and symbolic execution use to measure code coverage are sometimes a slight variations of the ones mentioned earlier.

Fuzzing: The term fuzzing refers to a family of automated input generating techniques that are widely used in the industry to find vulnerabilities and bugs in all kinds of software [35]. In general, a fuzzer aims at discovering inputs that generate new behaviors, thus one measure of success for fuzzer is code coverage. Simple statement coverage is rarely a good choice, since crashes do not usually depend on a single program statement, but on a specific sequence of statements [39]. Most fuzzing algorithms choose to define their own code coverage metric. American Fuzzy Lop (AFL) is a state of the art fuzzer that has seen extensive use in the industry in its base form, while new fuzzers are continuously built on top of it [32]. The measure used by AFL for code coverage lays between path and count-path coverage as it approximates the number of times that edges are traversed by specified intervals of natural numbers ($[1]$, $[2]$, $[3]$, $[4-7]$, $[8-15]$, $[16-31]$, $[32-127]$, $[128,\infty]$). Libfuzzer [30] and honggfuzz [36] employ count-statement coverage. To the best our our knowledge trace coverage is never used as it is infeasible in practice [16].

Dynamic Symbolic Execution: DSE is a well known dynamic analysis technique that combines concrete and symbolic execution [22]. DSE typically starts by executing a program on a random input and then generates branch conditions that take into account the executed branches. When execution ends, DSE looks at the last branch condition generated and uses a theorem prover to solve the negated predicate in order to explore the branch that was not executed. This is akin to symbolic execution, but DSE can use the concrete values obtained in the execution to simplify the job of the theorem prover. The ideal goal of DSE is to reach path coverage, which is always guaranteed if the conditions in the target program only contain linear arithmetics [22]. Thus, the efficacy of DSE in generating new inputs is measured in terms of path coverage formalised as $\mathcal{R}_{CFG}$ in our framework.

Let us denote with $\mathcal{R} \in eq(\Sigma^*)$ the equivalence relation modelling the code coverage metric used either by fuzzing or symbolic execution or any other algorithm for input generation. When $Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t})$ covers P wrt $\mathcal{R}$, we have that the fuzzer or symbolic execution algorithm has found all the inputs that allow us to observe the different behaviours of P wrt $\mathcal{R}$. In general, a dynamic analysis may be interested in a property $\mathcal{R}_A \in eq(\Sigma^*)$ that is different from the property $\mathcal{R}$ used to measure code coverage. When $\mathcal{R} \sqsubseteq \mathcal{R}_A$ we have that if $Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t})$ covers P wrt $\mathcal{R}$, then $Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t})$ covers P also wrt $\mathcal{R}_A$ and this means that the code coverage metric $\mathcal{R}$ can help in limiting the number of false negative of the dynamic analysis $\langle \mathcal{R}_A, Exe(\mathsf{P}, \mathsf{T}_\mathsf{P}, \mathsf{t})\rangle$. When $\mathcal{R} \not\sqsubseteq \mathcal{R}_A$ then a different metric for code coverage should be used (for example $\mathcal{R}_A$ itself).

4.3 Harming Dynamic Data Analysis

Data obfuscation transformations change the representation of data with the aim of hiding both variable content and usage. Usually, data obfuscation requires the program code to be modified, so that the original data representation can be reconstructed at runtime. Data obfuscation is often achieved through data encoding [5, 28]. More specifically, in [15, 23] data encoding for a variable x is formalised as a pair of statements: encoding statement $\mathsf{C}_{enc} = \mathsf{x} := \mathsf{f}(\mathsf{x})$ and decoding statement $\mathsf{C}_{dec} = \mathsf{x} := \mathsf{g}(\mathsf{x})$ for some function f and g, such that $\mathsf{C}_{dec}; \mathsf{C}_{enc} = \mathit{skip}$. According to [15, 23] a program transformation $\mathcal{T}(\mathsf{P}) \overset{\text{def}}{=} \mathsf{C}_{dec}; \mathsf{t}_\mathsf{x}(\mathsf{P}); \mathsf{C}_{enc}$ is a data obfuscation for x where t_x adjusts the computations involving x in order to preserve program's functionality, namely $Den[\![\mathsf{P}]\!] = Den[\![\mathsf{C}_{dec}; \mathsf{t}_\mathsf{x}(\mathsf{P}); \mathsf{C}_{enc}]\!]$. In Fig. 5 we provide a simple example of data obfuscation from [15, 23] where $\mathsf{C}_{enc} = \mathsf{x} := 2 * \mathsf{x}$ and $\mathsf{C}_{dec} = \mathsf{x} := \mathsf{x}/2$ and $\mathcal{T}(\mathsf{P}) = \mathsf{x} := \mathsf{x}/2; \mathsf{t}_\mathsf{x}(\mathsf{P}); \mathsf{x} := 2 * \mathsf{x}$ and program P is the one considered in Example. 2. This data transformation induces imprecision in the static analysis of the possible values assumed by x at program point $\bullet$. Indeed, the static analysis of the interval of values of x at program point $\bullet$ in $\mathcal{T}(\mathsf{P})$ is different and wider (it contains spurious values) than the interval of possible values of x at $\bullet$ in P. However, the dynamic analysis of properties on the values assumed by x during execution at the different program points

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

$\mathcal{T}(\mathsf{P})$
```
input x;
x := 2*x;
sum := 0;
while x < 2*50
• X = [x, 2 * 50 − 1]
   sum := sum + x/2;
   x := x + 2;
x:= x/2;
```

$\mathcal{T}_n(\mathsf{P})$
```
input x;
x := n*x;
sum := 0;
while x < n*50
• X = [x, n * 50 − 1]
   sum := sum + x/n;
   x := x + n;
x:= x/n;
```

$\mathcal{T}_H(\mathsf{P})$
```
input x;
n := select(N,x);
x := H^e(n,x);
sum := H^e(n,0);
while x <_H H^e(n,50)
• X = [x, H^e(n, 50) − 1]
   sum := sum +_H x;
   x := x +_H H^e(n,1);
x:= H^d(x);
```

Fig. 5: From the left: programs P, $\mathcal{T}(\mathsf{P})$, $\mathcal{T}_n(\mathsf{P})$ and $\mathcal{T}_H(\mathsf{P})$

(e.g., maximal/minimal value, number of possible values, interval of possible values) has not been hardened in $\mathcal{T}(\mathsf{P})$. The values assumed by x at $\bullet$ in $\mathcal{T}(\mathsf{P})$ are different from the values assumed by x at $\bullet$ in P but these properties on the values assumed by x are precisely observable by dynamic analysis on $\mathcal{T}(\mathsf{P})$. Transformation $\mathcal{T}(\mathsf{P})$ changes the properties of data values wrt P, but it does it in an invariant way: during every execution of $\mathcal{T}(\mathsf{P})$ we have that x is iteratively incremented by 2 and the guard of the loop becomes $x < 2 * 50 - 1$, and this is observable on any execution of $\mathcal{T}(\mathsf{P})$. This means that by dynamic analysis we could learn that the maximal value assumed by x is $99 (= 2 * 50 - 1)$. Thus, transformation $\mathcal{T}$ is not potent wrt properties of data values according to Definition 6 since it does not diversify the properties of values assumed by variables among traces. In order to hamper dynamic analysis we need to diversify data among traces, thus forcing dynamic analysis to observe more execution traces to be sound. We could do this by making the encoding and decoding statements parametric on some natural number n as described by the third program $\mathcal{T}_n(\mathsf{P}) = x := x/n; t_{x,n}(\mathsf{P}); x := n * x$ in Fig. 5 (which is the same as Q in Example. 2). Indeed, the parametric transformation $\mathcal{T}_n(\mathsf{P})$ is potent wrt properties that observe data values since it diversifies the values assumed by x among different executions thanks to the parameter n. For example, to observe the maximal value assumed by x in $\mathcal{T}_n(\mathsf{P})$ we should observe an execution for every possible value of n.

This confirms what observed [28]: existing data obfuscation makes static analysis imprecise but it is less effective against dynamic analysis. Interpreting data obfuscation in our framework allows us to see that, in order to hamper dynamic analysis, data encoding needs to diversify among traces. This can be done by making the existing data encoding techniques parametric.

Homomorphic encryption: As argued above, in order to preserve program functionality the original program code needs to be adapted to the encoding. In general, automatically deriving the modified code $t_x(\mathsf{P})$ for a given encoding on every possible program may be complicated. In this setting, an ideal situation is the one provided by fully homomorphic encryption where any operation on the original data has its respective for the encrypted data. It has been proven that fully homomorphic encryption is possible on any circuit [17]. Let H^e and H^d be the fully homomorphic encryption and decryption procedures. We could design a data obfuscation for the variables in P as $\mathsf{H}^d; \mathsf{P}_\mathsf{H}; \mathsf{H}^e$ where the program variables are encrypted with H^e, the computation is carried on the encrypted values by using homomorphic operations (denoted with subscript $_\mathsf{H}$), and at the end the final values of the variables are decrypted with H^d. Thus, the original program P and P_H are exactly the same programs where the operations have been replaced by their homomorphic version. In Fig. 5 on the right we show how a homomorphic encoding of program P would work, where the subscript $_\mathsf{H}$ denotes the homomorphic operations on the encrypted values. Encryption and decryption procedures have a random nature and use a key (that may be dependent on input values). Thus, the values of encrypted data varies among program traces. Moreover, since successive encryptions of the same values would lead to different encrypted values, we have that re-runs on the same values would generate different encrypted values. This proves that homomorphic encryption could be useful to design a potent data obfuscation against dynamic analysis: as it can diversify the encrypted data values among traces and the original values are retrieved only at the end of the computation.

Abstract Software Watermarking: In [12] the authors propose a sophisticated software watermarking algorithm and prove its resilience against static program analysis. The watermark can be extracted only by analysing the program on a specific congruence domain that acts like a secret key. The authors discuss some possible countermeasures against dynamic analysis that could reveal the existence of the watermark (and then remove it). Interestingly, the common idea behind these countermeasures is diversification of the property of data values that the dynamic analyses observe.

5 Related Works

To the best of our knowledge we are the first to propose a formal framework for dynamic analysis efficacy based on semantic properties. Other works have proposed more empirical ways to assess the impact of dynamic analysis.

Evaluating Reverse Engineering. Program comprehension guided by dynamic analysis has been evaluated with specific test cases, quantitative measures and the involvement of human subjects [8]. For example, comparing the effectiveness of static analysis and dynamic analysis towards the feature location task has been carried out through experiments involving 2 teams of analysts solving the same problem with a static analysis and a dynamic analysis approach respectively [37]. In order to compare the effectiveness of different reverse engineering techniques (which often employ dynamic analysis), Sim et al. propose the use of common benchmarks [34]. The efficacy of protections against human subjects has been evaluated in a set of experiments by Ceccato et al., finding that program executions are important to understand the behavior of obfuscated code [4]. Our approach characterizes dynamic attacks and protections according to their semantic properties which is an orthogonal work that can be complemented by more empirical approaches.

Obfuscations Against Dynamic Analysis. One of the first works tackling obfuscations specifically geared towards dynamic analysis is by Schrittwieser and Katzenbeisser [27]. Their approach adopts some principles of software diversification in order to generate additional paths in the CFG that are dependent on program input (i.e. they do not work for other inputs). Similar to this approach, Pawlowski et al. [26] generate additional branches in the CFG but add non-determinism in order to decide the executed path at runtime. Both of these works empirically evaluate their methodology and classify it with potency and resilience, two metrics introduced by Collberg et al. [6]. Banescu et al. empirically evaluated some obfuscations against dynamic symbolic execution (DSE) [2], finding that DSE does not suffer from the addition of opaque branches since they do not depend on program input. To overcome this limitation they propose the Range Dividers obfuscation that we illustrated in Section 4. A recent work by Ollivier et al. refines the evaluation of protections against dynamic symbolic execution with a framework that enables the optimal design of such protections [25]. All these works share with us the intuition that dynamic analysis suffers from insufficient path exploration and they prove this intuition with extensive experimentation. Our work aims at enabling the formal study of these approaches.

Formal Systems. Dynamic taint analysis has been formalized by making explicit the taint information in the program semantics [29]. Their work focuses on writing correct algorithms and shows some possible pitfalls of the various approaches. Ochoa et al. [24] use game theory to quantify and compare the effectiveness of different probabilistic countermeasures with respect to remote attacks that exploit memory-safety vulnerabilities. In our work we model MATE attacks. Shu et al. introduce a framework that formalizes the detection capability in existing anomaly detection methods [33]. Their approach equates the detection capability to the expressiveness of the language used to characterize normal program traces.

6 Discussion and Future Works

This work represents the first step towards a formal investigation of the precision of dynamic analysis in relation with dynamic code attack and defences. The results that we have obtained so far confirm the initial intuition: *diversification is the key for harming dynamic analysis*. Dynamic analysis generalises what it learns from a partial observation of program behaviour, diversification makes this generalisation less precise (dynamic analysis cannot consider what it has not observed). We think that this work would be the basis for further interesting investigations. Indeed, there are many aspects that still need to be understood for the development of a complete framework for the formal specification of the precision of dynamic analysis (no false negatives), and for the systematic development of program transformations that induce imprecision.

We plan to consider more sophisticated properties than the ones that can be expressed as equivalence relations. It would be interesting to generalise the proposed framework wrt to any semantic property that can be formalised as a closure operator on trace semantics. The properties that we have considered so far correspond to the set of atomistic closures where the abstract domain is additive. We would like to generalise our framework to properties modelled as abstract domains and where the precision of dynamic analysis is probably characterised in terms of the join-irreducible elements of such domains. A further investigation would probably lead to a classification of the properties usually considered by dynamic analysis: properties of traces, properties of sets of traces, relational properties, hyper-properties, together with a specific characterisation of the precision of the analysis and of the program transformations that can reduce it. This unifying framework would provide a common ground where to interpret and compare the potency of different software protection techniques in harming dynamic analysis.

We can view dynamic analysis as a learner that observes properties of some execution traces (training set) and then generalises what it has observed, where the generalisation process is the identity function. We wonder what would happen if we consider more sophisticated generalisation processes such as the ones used by machine learning. Would it be possible to define what is learnable? Would it be possible to formally define robustness in the adversarial setting? We think that this is an intriguing research direction and we plan to pursue it.

Acknowledgments The research has been partially supported by the project "Dipartimenti di Eccellenza 2018-2022" funded by the Italian Ministry of Education, Universities and Research (MIUR).

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. 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.
4. Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, 2014.
5. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.
6. 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.
7. 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.
8. Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.
9. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, 1977.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, 1979.
12. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185. ACM Press, New York, NY, 2004.
13. M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.
14. Mila Dalla Preda and Isabella Mastroeni. Characterizing a property-driven obfuscation strategy. *Journal of Computer Security*, 26(1):31–69, 2018.

15. S. Drape, C. Thomborson, and A. Majumdar. Specifying imperative data obfuscations. In *ISC - Information Security*, volume 4779 of *Lecture Notes in Computer Science*, pages 299 – 314. Springer Verlag, 2007.
16. Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.
17. Craig Gentry and Dan Boneh. *A fully homomorphic encryption scheme*, volume 20. Stanford university Stanford, 2009.
18. 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.
19. R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In O. Kiselyov and S. Thompson, editors, *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, pages 63 – 72. ACM Press, 2012.
20. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.
21. Roberto Giacobazzi, Isabella Mastroeni, and Mila Dalla Preda. Maximal incompleteness as obfuscation potency. *Formal Asp. Comput.*, 29(1):3–31, 2017.
22. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.
23. 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.
24. Martín Ochoa, Sebastian Banescu, Cynthia Disenfeld, Gilles Barthe, and Vijay Ganesh. Reasoning about probabilistic defense mechanisms against remote attacks. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 499–513. IEEE, 2017.
25. 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.
26. 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.
27. Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *International workshop on information hiding*, pages 270–284. Springer, 2011.
28. 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.
29. Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.
30. Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.
31. 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.

32. Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.
33. Xiaokui Shu, Danfeng Daphne Yao, and Barbara G Ryder. A formal framework for program anomaly detection. In *International Symposium on Recent Advances in Intrusion Detection*, pages 270–292. Springer, 2015.
34. Susan Elliott Sim, Steve Easterbrook, and Richard C Holt. Using benchmarking to advance research: A challenge to software engineering. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 74–83. IEEE, 2003.
35. Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
36. Robert Swiecki. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz*, 2016.
37. Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.
38. 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.
39. Michal Zalewski. Technical" whitepaper" for afl-fuzz. *URl: http://lcamtuf. coredump. cx/afl/technical_details. txt*, 2014.