Mila Dalla Preda

Code Obfuscation and Malware Detection by Abstract Interpretation

Ph.D. Thesis

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Roberto Giacobazzi

Series N°: **TD ????**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Summary

An obfuscating transformation aims at confusing a program in order to make it more difficult to understand while preserving its functionality. Software protection and malware detection are two major applications of code obfuscation. Software developers use code obfuscation in order to defend their programs against attacks to the intellectual property, usually called malicious host attacks. In fact, by making the programs more difficult to understand it is possible to obstruct malicious reverse engineering – a typical attack to the intellectual property of programs. On the other side, malware writers usually obfuscate their malicious code in order to avoid detection. In this setting, the ability of code obfuscation to foil most of the existing detection techniques, such as misuse detection algorithms, relies in their purely syntactic nature that makes malware detection sensitive to slight modifications of programs syntax. In the software protection scenario, researchers try to develop sophisticated obfuscating techniques that are able to resist as many attacks as possible. In the malware detection scenario, on the other hand, it is important to design advanced detection algorithms in order to detect as many variants of obfuscated malware as possible. It is clear how both malicious host and malicious code attacks represent harmful threats to the security of computer networks.

In this dissertation, we are interested in both security issues described above. In particular, we describe a formal approach to code obfuscation and malware detection based on program semantics and abstract interpretation. This theoretical framework is useful in contrasting some well known drawbacks of software protection through code obfuscation, as well as for improving existing malware detection schemes. In fact, the lack of rigorous theoretical bases for code obfuscation prevents any possibility to formally study and certify their effectiveness in protecting proprietary programs. Moreover, in order to design malware detection schemes that are resilient to obfuscation we have to focus on program semantics rather than on program syntax.

Our formal framework for code obfuscation relies on a semantics-based definition of code obfuscation that characterizes each program transformation T as a potential obfuscation in terms of the most concrete property preserved by T on program semantics. Deobfuscating techniques, and reverse engineering in general, usually begin with some sort of static program analysis, which can be specified as an abstraction of program semantics. In the software protection scenario, this observation naturally leads to model attackers as abstractions of program semantics. In fact, the abstraction modeling the attacker expresses the amount of information, namely the semantic properties, that the attacker is able to observe. It follows that, comparing the degree of abstraction of an attacker A with the one of the most concrete property preserved by an obfuscating transformation T, it is possible to understand whether obfuscation T defeats attack A. Following the same reasoning it is possible to compare the efficiency of different obfuscating transformations, as well as the ability of different attackers in contrasting a given obfuscation. We apply our semantics-based framework to a known control code obfuscation technique that aims at confusing the control flow of the original program by inserting opaque predicates.

As argued above, an obfuscating transformation modifies a program while preserving an abstraction of its semantics. This means that different obfuscated versions of the same malware have to share (at least) the malicious intent, namely the maliciousness of their semantics, even if they may express it through different syntactic forms. The basic idea of our formal approach to malware detection is to use program semantics to model both malware and program behaviour, and semantic abstractions to hide the details changed by the obfuscation. Thus, given an obfuscation T, we are interested in defining an abstraction of program semantics that does not distinguish between the semantics of malware M and the semantics of its obfuscated version $T(M)$. In particular, we provide this suitable abstraction for an interesting class of commonly used obfuscating transformations. It is clear that, given a malware detector D, it is always possible to define its semantic counterpart by analyzing how D works on program semantics. At this point, by translating both malware detectors and obfuscating transformations in the semantic world, we are able to certify which obfuscations a detector is able to handle. This means that our semantics-based framework provides a formal setting where malware detectors designers can prove the efficiency of their algorithms.

Acknowledgements

The first person that I would like to thank is my advisor Roberto Giacobazzi for his precious guide and encouragement over these years. He taught me how to develop my ideas and how to be independent.

A great thanks goes to Saumya Debray for his very kind hospitality and constant support while I was visiting the department of Computer Science at Tucson. I sincerely thank Somesh Jha and Mihai Christodorescu for the interesting discussions we had and their precious collaboration. I also thank Matias Madou and Koen De Boscheere for the work done together.

A warm thank goes to the participants of the Doctoral Symposium affiliated to Formal Methods 2006 and in particular to the organizers Ana Cavalcanti, Augusto Sampaio and Jim Woodcock for their interesting comments and advices on my work.

I would also like to thank my PhD thesis referees Christian Collberg and Patrick Cousot, but also Andrea Masini and Massimo Merro for their precious advices and comments on my studies.

Contents

Preface

This thesis is composed by seven chapters. Each chapter provides a brief introduction explaining its contents, while the chapters describing original work have also a final discussion section about the problems addressed and the solutions proposed in the chapter.

Much of the content of this thesis has already been published. In particular, Chapter 4 was developed together with Roberto Giacobazzi [48], Chapter 5 presents the results obtained in two related works one in collaboration with Roberto Giacobazzi [47] and the other one with Roberto Giacobazzi, Koen De Bosschere and Matias Madou [49], while Chapter 6 is based on a recent joint work with Mihai Christodorescu, Saumya Debray and Somesh Jha [46] that was developed during my visit at the Department of Computer Science of the University of Arizona where I've joint the research group of Saumya Debray.

The work of this thesis focuses on code obfuscation – a program transformation that is commonly used by software developers to protect the intellectual property of their programs and by malicious code writers to avoid detection. In the first scenario we are interested in the design of powerful obfuscating techniques, while in the second one in the design of advanced tools for defeating obfuscation. The work presented in this thesis clearly represents the dual nature of the main application fields of code obfuscation.

1

Introduction

The widespread development of computer networks and Internet technology gave rise to new computational frameworks. If, on the one hand, remote execution, distributed computing and code mobility add flexibility and new computing abilities, on the other hand they raise security and safety problems that were not an issue when computation was carried out on stand-alone machines. Hosts and networks must be protected from malicious agents (programs) and agents (programs) must be protected from malicious hosts. A key concern for software developers is to defend their programs against *malicious host* attacks, that usually aim at stealing, modifying or tampering with the code in order to take (economic) advantages over it. Besides, a related security issue involves the execution of *malicious code* on a host machine. A malicious program may try to gain privileged or unauthorized access to resources or private information, or may attempt to damage the machine on which it is executed (e.g., computer viruses). Both malicious host and malicious code attacks represent harmful threats to the security of computer networks.

1.1 Motivations

The Malicious Host Perspective

Malicious reverse engineering, software piracy and software tampering are the most common attacks against proprietary programs [33]. Given a software application the aim of reverse engineering is to analyze it in order to understand its inner working and acquire the knowledge needed to redesign the program. Observe that reverse engineering can be used also for benign purposes, as for example by software developers to improve their own products. The difficulty of software reverse engineering relies in reconstructing enough knowledge about a program in order to modify it, reuse parts of it or interface with it. It is clear that this knowledge can be used for unlawful purposes – this is called malicious

reverse engineering. In fact, programmers may reduce both cost and time of software development by extracting proprietary algorithms and data structures from a rival application, and reuse these parts in their own products. Obviously, this kind of attacks violate the intellectual property of software. Observe that both software tampering and software piracy need a preliminary reverse engineering phase in order to understand the inner working of the program they want to tamper with or to steal. Thus, *preventing malicious reverse engineering is a crucial issue when defending programs against malicious host attacks.* A number of legal and technical methods to protect the intellectual property of software exists. While legal defenses are usually expensive, and therefore prohibitive for small companies, technical methods are generally cheaper and may represent a more attractive solution. In particular, making reverse engineering so difficult to be impractical, ideally impossible, is a common goal of many technical approaches to software protection. These defense techniques include the use of hardware devices, server-side execution, encryption, and obfuscation. Hardware devices protect a program by relating its execution to the presence of certain hardware features (e.g., a dongle). However, hardware devices do not provide a complete solution to the malicious host problem, and their employment usually meets stiff resistance from users. Server-side execution techniques prevent the malicious host from having physical access to the program by running it remotely, and it is therefore sensitive to performance degradation due to network communication. Program encryption works only if the encryption/decryption process takes place in hardware and therefore suffers from the same limitations as hardware devices. Code obfuscation techniques aim at transforming programs in order to make them difficult to understand and analyze, while preserving their functionality. Code obfuscation is a low cost technique that does not affect portability and it represents one of the most promising methodologies for defending mobile programs against malicious host attacks. This is witnessed by the increasing interest in this technology, which, in recent years, has lead to the design of many obfuscating transformations (e.g., [29,31,33,35,100,123,148]).

The Malicious Code Perspective

Malicious programs are usually classified according to the type of damage they perform and the methodology they use to spread (e.g., viruses, worms, Trojan horses) [110]. In general, the term malware is used to refer to a malicious code regardless of classification. In fact, a malware is defined to be a program with a malicious intent designed to propagate with no user consent and to damage the machine over which it is executed or the network over which it communicates. For example, a piece of malware may be designed to gain unauthorized access to sensitive information in order to tamper with it, delete it or communicate it to a

third party with malicious intent. One major cause of the widespread use of malicious code is the global connectivity of computers through the Internet, that makes machines vulnerable to attacks from remote sources and increases the speed of malware infection. The growth in the complexity of modern computing systems makes it difficult, if not impossible, to avoid bugs. This increments the possibility of malicious code attacks, that usually exploit such vulnerabilities in order to damage the systems. Moreover, it is easier to mask or hide malicious code in complex and sophisticated systems. In fact, when the size and complexity of a system grow it becomes more difficult to analyze it in order to prove that it is not infected. Thus, the threat of malicious code attacks is an unavoidable problem in computer security, and therefore *it is crucial to detect the presence of malicious code in software systems*. A considerable body of literature on techniques for malware detection exists – Szor provides an excellent summary [140]. In particular, two major approaches to malware detection are misuse and anomaly detection. Misuse detection, also called signature-based detection, classifies a program P as infected by a malware when the malware signature – a sequence of instructions characterizing the malware – occurs in P. In general, signature-based algorithms detect already known malware, while they are ineffective against unknown malicious programs, since no signature is available for them. On the other side, anomaly detection algorithms are based on a notion of normal program behaviour and classify as malicious any behaviour deviating from normality. In general, machine learning techniques and statistical methods are used to define normal behaviours, which turn out to be a quite hard task. It is clear that anomaly detection does not need any a priori knowledge of the malicious code, and can therefore handle previously unseen malware. As a drawback, this technique generally produces many false alarms, since systems often exhibit unseen or unusual behaviours that are not malicious. Thus, misuse detection and anomaly detection techniques have advantages that complement each other, together with limitations with no clear solution up to now [111].

1.2 The Problem

In this dissertation we are interested in both aspects of computer security: the malicious host perspective, concerning the protection of the intellectual property of a proprietary program running on a malicious host, and the malicious code view related to the defense of hosts against malware attacks.

The Malicious Host Perspective

As observed above code obfuscation provides a promising technical approach for protecting the intellectual property of software. However, *the lack of a rigorous*

theoretical background is a major drawback of code obfuscation. The absence of a theoretical basis makes it difficult to formally analyze and certify the effectiveness of such obfuscating techniques in contrasting malicious host attacks. Therefore, it is hard to compare different obfuscating transformations with respect to their resilience to attacks, making it difficult to understand which technique is better to use in a given scenario. Few theoretical works on code obfuscation exist, so that the design of a formal framework where modeling, studying and relating obfuscating transformations and attacks is still in an early stage. Thus, it is not surprising that different definitions of code obfuscation exist, some of them have led to promising results, while other have led to impossibility results. For example, the positive theoretical results by Wang et al. [147, 148] showing the NP-hardness of a specific code obfuscation technique, the related ones by Ogiso et al. [123], and the PSPACE-hardness result by Chow et al. [22], provide evidence that code obfuscation can be an effective technique for preventing malicious reverse engineering. By contrast, a well known negative theoretical result by Barak et al. [11] shows that, according to their formalization of obfuscation, code obfuscation is impossible. At a first glance, this result seems to prevent code obfuscation at all. However, this result is stated and proved in the context of a rather specific and ideal model of code obfuscation. Given a program P, Barak et al. [11] define an obfuscator as a program transformer $\mathcal{O}$ that satisfies the following conditions: (1) $\mathcal{O}(P)$ is functionally equivalent to P, (2) the slowdown of $\mathcal{O}(P)$ with respect to P is polynomial both in time and space, and (3) anything that one can compute from $\mathcal{O}(P)$ can also be computed from the input-output behaviour of P. This formalizes an "ideal" obfuscator, while in practice these constraints are commonly relaxed. For example, in [33–35,124,148] the authors allow the obfuscated programs to be significantly slower or larger than the original ones, or to have different side-effects. In fact, according to a standard definition, an obfuscator is a potent program transformation that preserves the observational program behaviour, namely the behaviour experimented by the user. Here, potent means that the transformed program is more complex, i.e., more difficult to reverse engineer, than the original one [31,34,35]. Consequently, the notion of code obfuscation is based on a fixed metric for program complexity, which is usually defined in terms of syntactic program features, such as code length, number of nesting levels, numbers of branching instructions, etc. [34]. Complexity measures based on program semantics are instead less common, even if they may provide a deeper insight in the real potency of code obfuscation. In fact, if, on the one hand, code obfuscation aims at confusing some (usually syntactic) information, on the other hand it has to preserve program behaviour (namely program semantics to some extent).

The Malicious Code Perspective

In the malware detection scenario, we focus on signature-based algorithms that are widely used thanks to their low false positive rate and ease of use. In order to deal with advanced detection systems, malware writers, viz. hackers, recur to sophisticated hiding techniques. This parallel evolution of defense and attack techniques have led to the development of smart malware, as for example the so-called metamorphic malware. The basic idea of metamorphism is that each successive generation of a malware changes the syntax while leaving the semantics unchanged in order to foil misuse detection systems. In this setting, code obfuscation may be used to syntactically transform a malware, and therefore its signature, while maintaining its functional behaviour, namely its malicious intent. In fact, code obfuscation turns out to be one of the most powerful countermeasures used by hackers against signature-based detection algorithms. Recent results [24] show that signature-based algorithms can be defeated using simple obfuscating techniques, including code transposition, semantic NOP insertion, substitution of equivalent instruction sequences, opaque predicate insertion and variable renaming. These results provide strong evidence that signature matching methodologies are not resilient to slight modifications of malware and that they need a frequently updated database of malware signatures, i.e., one for each version of the malware. Therefore, *an important requirement for a robust malware detection technique is the capability of handling obfuscating transformations*. The reason why obfuscation can easily foil signature matching lies in the syntactic nature of this approach that ignores program functionality. In fact, code obfuscation changes the malware syntax but not its intended behaviour, which has to be preserved. Formal methods for program analysis, such as semantics-based static analysis and model checking, could be useful in designing more sophisticated malware detection algorithms that are able to deal with obfuscated versions of the same malware. For example, Christodorescu et al. [25] put forward a semantics-aware malware detector that is able to handle some of the obfuscations commonly used by hackers, while Kinder et al. [84] introduce an extension of the CTL temporal logic, which is able to express some malicious properties that can be used to detect malware through standard model checking algorithms. These preliminary works confirm the potential benefits of a formal approach to malware detection.

Since hackers frequently recur to code obfuscation in order to avoid detection, one major criterion for evaluating a new malware detection algorithm is its resilience to obfuscation. In general, the identification of the set of obfuscations that a malware detector can handle is a complex and error-prone task. The main difficulty comes from the fact that there exists a large number of obfuscating techniques developed both by hackers and software developers. Moreover, specific techniques can always be introduced in order to foil a particular detection

scheme. Further difficulties are related to the fact that detectors and obfuscating transformations are commonly defined using different languages (e.g., program analysis vs program transformation). Thus, in order to certify the efficiency of malware detection algorithms *a formal framework where to prove the resilience of a malware detector scheme against classes of obfuscating transformations would be useful.*

1.3 The Idea: A Semantics-based Approach

The Malicious Host Perspective

As argued above, obfuscating transformations change how programs are written while preserving their functional behaviour, namely their semantics. In order to formalize and quantify the amount of "obscurity" added by an obfuscating transformation, namely how much more complex the transformed program is to reverse engineer with respect to the original one, we need a formal model for obfuscation as well as for attackers, i.e., code deobfuscation. Deobfuscating techniques, and reverse engineering in general, usually begin with some sort of static program analysis. Recently, it has been shown how the combination of static and dynamic analyses may lead to powerful deobfuscating tools [144]. If, on the one hand, a static program analysis can be specified as an abstract interpretation, i.e., an approximation, of a concrete program semantics [41], on the other hand a dynamic analysis can be seen as a possibly non-decidable approximation of a concrete program semantics. This observation suggests that attackers may be modeled as abstraction of concrete program semantics and confirms the potential benefits that may originate from the introduction of semantics-based metrics for program complexity. In fact, measuring the differences between the original and the obfuscated program in terms of their semantics provides a better insight on what the transformation really hides, and therefore on what an attacker is able to observe and deduce from the obfuscated code. Program semantics precisely formalizes the meaning of a program, namely its behaviour, and it is not sensitive to minor changes in program syntax, namely how a program is written. The idea is to address code obfuscation from a semantic point of view, by considering the effects that obfuscating transformations have on program semantics. Recall that program semantics formalizes the behaviour of a program for every possible input, and that the precision of such description depends on the level of abstraction of the considered semantics, namely on the precision of the domain over which the semantics is defined. In particular, Cousot [40] defines a hierarchy of semantics, where semantics at different levels of abstractions are specified as successive approximations of a given concrete semantics, namely trace semantics. In the following, concrete program semantics refers to trace semantics, that observes step by step the history of each possible computation, while abstract

semantics refers to any abstraction of trace semantics. Note that the semantics modelling the input-output (observational) behaviour of a program, being an abstraction of trace semantics, is an element in this hierarchy.

A recent result by Cousot and Cousot [44] formalizes the relation between syntactic and semantic transformations within abstract interpretation, where programs are seen as abstractions of their semantics. In this setting, abstract interpretation theory provides the right framework in which to relate syntactic and semantic transformations. In fact, according to well known results in abstract interpretation, given a concrete (semantic) transformation it is always possible to define its abstract (syntactic) counterpart and vice versa. Hence, this gives us the right tool for reasoning about the semantic aspects of obfuscating transformations, and for deriving new obfuscating techniques as approximations of semantic transformations of interest.

According to the standard definition of code obfuscation, the original and obfuscated program exhibit the same observational behaviour, meaning that obfuscation has to preserve an abstraction of program trace semantics. Reasoning on the semantic aspects of obfuscation, one is naturally led to *model obfuscating transformations in terms of the most concrete preserved semantic property, and attackers as abstractions of concrete program semantics.* In particular, we provide a theoretical framework, based on program semantics and abstract interpretation, where formalizing, studying and relating different obfuscating transformations with respect to their potency and resilience to attacks.

The Malicious Code Perspective

As argued above, a semantics-based approach to malware detection may be the key for improving existing detection algorithms. In fact, different obfuscated versions of the same malware have to share (at least) the malicious intent, namely the maliciousness of their semantics, even if they may express it through different syntactic forms. Our idea is to use program trace semantics to model both malware and program behaviours, and abstract interpretation to hide the details changed by obfuscation. In fact, it turns out that *the semantics of different obfuscated versions of the same malware have to be equivalent up to some abstraction.* Thus, given an obfuscation $\mathcal{O}$, we are interested in the abstract semantic property $\mathcal{A}$ that the semantics of a malware M shares with the semantics of its obfuscated version $\mathcal{O}(M)$. The knowledge of the semantic abstraction $\mathcal{A}$ allows us to characterize program infection in terms of $\mathcal{A}$. A malware detection algorithm that verifies infection following this semantic test is called a semantic malware detector. It is clear that, given an obfuscation $\mathcal{O}$, a crucial point of such an approach is the definition of a suitable abstraction $\mathcal{A}$. In fact, if $\mathcal{A}$ is too coarse then a lot of programs would be misclassified as infected while they are

not, i.e., we might have an high false positive rate, meaning that the proposed detection algorithm is not sound. On the other side, if $\mathcal{A}$ is too concrete, then the detection process is very sensitive to obfuscation and a lot of infected programs will be classified as malware free while they are not, i.e., we might have many false negatives, meaning that the detection algorithm is not complete. Thus, the efficiency of the proposed detection approach clearly depends on the chosen abstraction $\mathcal{A}$.

Given a general malware detector D, it is always possible to define its semantic counterpart by analyzing how D works on program semantics. Next, by translating both malware detectors and obfuscating transformations in the semantic world we are able to certify the family of obfuscations that the detector is able to handle. In this setting program semantics turns out to be the right tool for proving soundness and completeness of malware detection algorithms with respect to a given class of obfuscating transformations.

1.4 Main Results

The Malicious Host Perspective

We observed how attackers, i.e., static and dynamic analyzers, at different levels of precision can be naturally modeled as abstractions of concrete program semantics. In fact, the abstract domain of computation modeling an attacker precisely captures the amount of information that the attacker is able to deduce while observing a program, or, otherwise stated, the semantic properties in which the attacker is interested. Thus, a coarse abstraction models an attacker that observes simple semantic properties while finer abstractions, being closer to program concrete semantics, model attackers that are interested in the very details of computation. Our model allows us to compare attackers with respect to their degrees of precision. Moreover, we propose a formal definition of code obfuscation where obfuscating transformations are characterized by the most concrete property they preserve on program semantics. In particular, a program transformation T is a $\mathcal{Q}$-obfuscator, where $\mathcal{Q}$ is the most concrete property preserved by T on program semantics, namely the most precise information that the original and transformed program have in common. According to this definition, any program transformation can be seen as a code obfuscation where the most concrete preserved property precisely expresses what can still be known after obfuscation, despite syntactic modifications, and therefore what attackers can deduce from the obfuscated program. In order to characterize the obfuscating behaviour of each program transformation, we provide a systematic methodology for deriving the most concrete property preserved by a given transformation. This notion of obfuscation is clearly parametric on the most concrete preserved property, and the observational behaviour is just a particular instance

of this definition. In particular, we show that our semantics-based notion of code obfuscation is a generalization of the standard definition of obfuscation. Since semantic properties are modeled, as usual, by abstractions of trace semantics, it turns out that obfuscating transformations can be compared to each other with respect to the degree of abstraction of the most concrete property they preserve. Given a $\mathcal{Q}$-obfuscator, the more abstract $\mathcal{Q}$ is the more potent the obfuscation is, meaning that a lot of details of the original program have been lost during the obfuscation phase. On the other hand, when $\mathcal{Q}$ is close to the concrete program semantics, it turns out that few details have been hidden by the obfuscation.

The semantics-based definition of code obfuscation, together with the abstract interpretation-based model of attackers, turn out to be particularly useful when considering control code obfuscation by opaque predicate insertion. Here, the obfuscating transformation confuses the original control flow of programs by inserting "fake" conditional branches guarded by opaque predicates, i.e., predicates that always evaluate to a constant value. It is clear that an attacker A is able to defeat such an obfuscation when A is able to disclose the inserted opaque predicates. Modeling attackers as abstract domains allows us to prove that the degree of precision needed by an attacker to break an opaque predicate can be expressed as an abstract domain property, known as completeness in abstract interpretation. This result is particularly interesting because it provides a precise formalization of the amount of information needed by an attacker to break a given opaque predicate. Moreover, this allows us to compare different attackers with respect to their ability to break a given opaque predicate, and different opaque predicates, according to their resilience to attackers.

The Malicious Code Perspective

In order to define a suitable abstraction $\mathcal{A}$ that allows a semantic malware detector to deal with as many obfuscations as possible, we focused on the effects that obfuscating transformations may have on malware semantics in order to isolate a common pattern. This leads us to the definition of a particular class of obfuscating transformations, characterized by the fact that they cause minor changes on malware semantics. These transformations are called conservative, since the original malware semantics is somehow still present in the semantics of the obfuscated malware, even if the syntax of the two codes may be quite different. We show that most obfuscating transformations commonly used by malware writers are actually conservative, and that the property of being conservative is preserved by composition. For this class of obfuscating transformations we are able to provide a suitable abstraction $\mathcal{A}_C$ that yields a precise detection of conservative variants of malware. In particular, we prove that the semantic malware detector based on $\mathcal{A}_C$ is both sound and complete for the above mentioned class of conservative obfuscations.

On the other hand, non-conservative transformations deeply modify malware semantics and this explains why we are not able to find a common pattern for handling non-conservative transformations as a whole. In fact, in this case, it is necessary to define an ad-hoc abstraction for each non-conservative transformation. However, we provide some possible solutions for deriving the desired abstraction. As an example, we design an abstraction that is able to precisely detect the variants of a malware obtained through variable renaming, which is a well known non-conservative obfuscation.

Of course, malware writers combine different obfuscating techniques in order to evade misuse detection. Thus, we investigate the relationship occurring between the abstractions that are able to deal with single obfuscations and the abstraction that is needed to defeat their combinations. In particular, it turns out that, under certain assumptions, the ability to deal with "elementary" obfuscations allows us to handle also their combinations.

The proposed semantic model turns out to be quite flexible. In fact, since our detection technique is based on the definition of a suitable abstraction and since abstractions can be composed, it turns out that our methodology can be weakened in many different ways in order to fit specific situations. In particular, a deeper knowledge of a given malware allows us to further specify the detection algorithm with respect to that malware and therefore to handle a wider class of obfuscating transformations.

In order to show how our framework can be used to prove soundness and completeness of malware detectors, we consider the semantics-aware malware detector defined by Christodorescu et. al [25] and the well known signature matching algorithm. In particular, we are able to prove the completeness of the semantics-aware malware detector for certain obfuscating transformations (soundness was already proved in [25]), and we show that signature-based detection is generally sound but not complete, namely it is complete for a very restricted class of obfuscating transformations.

1.5 Overview of the Thesis

This thesis is structured as follows. Chapter 2 provides notation and the basic algebraic notions that we are going to use in the following of the thesis, together with a brief introduction to abstract interpretation. In particular we present the recent work of Cousot and Cousot [44], where abstract interpretation is applied to program transformation. In Chapter 3 we present both the major techniques for software protection and the most common algorithms form malware detection. In particular, we recall Collberg's taxonomy of obfuscating transformations [34] and the most important theoretical results achieved in this field [11, 123, 148]. Moreover, we discuss advantages and disadvantages of exist-

ing malware detection schemes together with the most sophisticated tricks used by malware to avoid detection – such as polymorphism and metamorphism.

In Chapter 4 we present our semantics-based approach to code obfuscation. We describe how code obfuscation can be defined in terms of the most concrete property it preserves on program semantics, and how attackers can be modeled as abstractions of concrete program semantics. Furthermore, we describe how the proposed semantic model allows us to compare the resilience of different obfuscating transformations to attackers. Studying the obfuscating behaviour of constant propagation, we provide an example of the fact that any program transformation can be seen as an obfuscation in the proposed semantic framework.

In Chapter 5 we focus on control code obfuscations based on opaque predicate insertion. We study the effects of this transformation on program semantics and we derive an iterative algorithm for opaque predicate insertion following the methodology proposed in [44]. We consider two classes of numerical opaque predicates widely used by existing tools for obfuscation, and we show that the ability of an attacker to disclose such predicates can be expressed as a completeness problem in the abstract interpretation field. Next, we propose an opaque predicate detection algorithm based on this theoretical result which has better performances than existing detection schemes.

In Chapter 6 we address the malware detection problem from a semantic point of view. We provide a semantics-based notion of malware infection and we show how abstract interpretation can be used to deal with obfuscated malware. We provide a classification of obfuscating transformations based on their effects on program semantics. In particular, a transformation is conservative if it preserves the structure of trace semantics, non-conservative otherwise. We provide a methodology for handling conservative obfuscations and we prove that most commonly used obfuscating transformations are conservative. Next, we discuss how to deal with non-conservative obfuscations. To conclude we use our semantics-based framework to prove the precision of some existing malware detection algorithms.

Chapter 7 sums up the major contributions of this thesis and briefly describes future works that we would like to explore.

2

Basic Notions

In this chapter, we introduce the basic algebraic notation that we are going to use in the thesis. In Section 2.1 we describe the mathematical background, recalling the basic notions of sets, functions and relations, followed by an overview of fixpoint theory [42,142]. Moreover, we give a brief presentation of lattice theory, recalling the basic algebraic ordered structures and the definitions of upper closure operators and Galois connections and we describe how these two notions are related to each other. Standard references for lattice theory are [50, 62, 67]. In Section 2.2 we introduce abstract interpretation [41, 43], characterizing abstract domains in terms of both Galois connections and upper closure operators. Moreover, we describe the properties of soundness and completeness of abstract domains with respect to a given function, and we recall the existence of a domain transformer that adds the minimal amount of information to a given abstract domain in order to make it complete [61]. In Section 2.3 we describe the recent application of abstract interpretation to program transformations, where programs are seen as abstractions of their semantics [44], together with the presentation of the syntax and semantics of a simple imperative language that we will use in the rest of the thesis.

2.1 Mathematical Background

2.1.1 Sets

A *set* is a collection of objects (or elements). We use the standard notation $x \in C$ to express the fact that x is an element of the set C, namely that x belongs to C. The cardinality of a set C represents the number of its elements and it is denoted as $|C|$. Let C and D be two sets. C is a subset of D, denoted $C \subseteq D$, if every element of C belongs to D. When $C \subseteq D$ and there exists at least one element of D that does not belong to C we say that C is properly contained in D, denoted $C \subset D$. Two sets C and D are equal, denoted $C = D$,

if C is a subset of D and viceversa, i.e., $C \subseteq D$ and $D \subseteq C$. Two sets C and D are different, denoted $C \neq D$, if there exists an element in C (in D) that does not belong to D (to C). Let $\varnothing$ denote the empty set, namely the set without any element. In this case, for every element x we have that $x \notin \varnothing$ and for every set C we have that $\varnothing \subseteq C$. The set $C \cup D$ of elements belonging to C or to D is called the *union* of C and D, and it is defined as $C \cup D \stackrel{\text{def}}{=} \{x \mid x \in C \vee x \in D\}$. The set $C \cap D$ containing the elements belonging both to C and D identifies the *intersection* of C and D, and it is defined as $C \cap D \stackrel{\text{def}}{=} \{x \mid x \in C \wedge x \in D\}$. Two sets C and D are *disjoint* if their intersection is the empty set, i.e., $C \cap D = \varnothing$. Let $C \setminus D$ denote the set of elements of C that do not belong to D, formally $C \setminus D \stackrel{\text{def}}{=} \{x \mid x \in C \wedge x \notin D\}$. The *powerset* $\wp(C)$ of a set C is defined as the set of all possible subsets of C: $\wp(C) \stackrel{\text{def}}{=} \{D \mid D \subseteq C\}$. Let C^* denote the set of finite sequences of elements of C, where a sequence is denoted as $x_1...x_n$ with $x_i \in C$ and $\epsilon \in C^*$ denotes the empty sequence.

Relations

Let us see how it is possible to establish a relation between elements of sets. Let x, y be two elements of a set C, we call *ordered pair* the element (x, y), such that $(x, y) \neq (y, x)$. This notion can be extended to the one of ordered n-tuple of n elements $x_1...x_n$, with $n \geq 2$, by $(...((x_1, x_2), x_3)...)$, denoted by $(x_1...x_n)$.

Definition 2.1. Given n sets $\{C_i\}_{1 \leq i \leq n}$. We define the *cartesian product* of the n sets C_i as the set of ordered n-tuple:

$$C_1 \times C_2 \times ... \times C_n \stackrel{\text{def}}{=} \left\{ \ (x_1...x_n) \,\middle|\, \forall i : 1 \leq i \leq n : x_i \in C_i \ \right\}$$

Let C^n, $n \in \mathbb{N}$ and $n \geq 1$, denote the n-th cartesian self product of C. Given two not empty sets C and D, any subset of the cartesian product $C \times D$ defines a *relation* between the elements of C and the elements of D. In particular, when $C = D$ any subset of $C \times C$ defines a *binary relation* on C. Given a relation $\mathcal{R}$ between C and D, i.e., $\mathcal{R} \subseteq C \times D$, and two elements $x \in C$ and $y \in D$, then $(x, y) \in \mathcal{R}$ and $x\mathcal{R}y$ are equivalent notations denoting that the pair (x, y) belongs to the relation $\mathcal{R}$, namely that x is in relation $\mathcal{R}$ with y. In the following we introduce two important classes of binary relations on a set C.

Definition 2.2. A binary relation $\mathcal{R}$ on a set C is an *equivalence relation* if $\mathcal{R}$ satisfies the following properties:

- reflexivity: $\forall x \in C : (x, x) \in \mathcal{R}$;
- symmetry: $\forall x, y \in C : (x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$;
- transitivity: $\forall x, y, z \in C : (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$.

Given a set C equipped with an equivalence relation $\mathcal{R}$, we consider for each element x of C the subset C_x of C containing all the elements $y \in C$ in equivalence relation with x, i.e., $C_x = \{y \in C \mid x\mathcal{R}y\}$. The sets C_x are called *equivalence classes* of C as regard relation $\mathcal{R}$, and they are usually denoted as $[x]_{\mathcal{R}}$ with $x \in C$.

Definition 2.3. A binary relation $\leq$ on a set C is a *partial order* on C if the following properties hold:

- reflexivity: $\forall x \in C : x \leq x$;
- antisymmetry: $\forall x, y \in C : x \leq y \wedge y \leq x \Rightarrow x = y$;
- transitivity: $\forall x, y, z \in C : x \leq y \wedge y \leq z \Rightarrow x \leq z$.

Functions

Let C and D be two sets. A *function* f from C to D is a relation between C and D such that for each $x \in C$ there exists exactly one $y \in D$ such that $(x, y) \in f$ and in this case we write $f(x) = y$. Usually the notation $f : C \to D$ is used to denote a function from C to D, where C is the *domain* and D the *co-domain* of function f. The set $f(X) \stackrel{\text{def}}{=} \{f(x) \mid x \in X\}$ is the *image* of $X \subseteq C$ under f. In particular, the image of the domain, i.e., $f(C)$, is called the *range* of f. The set $f^{-1}(X) \stackrel{\text{def}}{=} \{y \in C \mid f(y) \in X\}$ is called the *reverse image* of $X \subseteq D$ under f. If there exists an elements $x \in C$ such that the element $f(x)$ is not defined, we say that function f is *partial*, otherwise function f is said to be *total*. Let us recall some basic properties of functions.

Definition 2.4. Given two sets C and D and function $f : C \to D$, we have that:

- function f is *injective* or *one-to-one* if for every $x_1, x_2 \in C : f(x_1) = f(x_2) \Rightarrow x_1 = x_2$;
- function f is *surjective* or *onto* if: $f(C) = D$;
- function f is *bijective* if f is both injective and surjective.

Thus, a function is injective if it maps distinct elements into distinct elements, while a function is surjective if every element of the co-domain is image of at least one element of the domain. Two sets are *isomorphic*, denoted $\cong$, if there exists a bijection between them. An interesting function is the *identity* function $id : C \to C$ that associates each element to itself, i.e., $\forall x \in C : id(x) = x$. The *composition* $g \circ f : C \to E$ of two functions $f : C \to D$ and $g : D \to E$, is defined as $g \circ f(x) \stackrel{\text{def}}{=} g(f(x))$. When it is clear from the context the symbol $\circ$ may be omitted and the composition can simply be denoted as gf. Sometimes, function f on variable x is denoted as $\lambda x.f(x)$. If $f : X^n \to Y$ is an n-ary function then its pointwise extension $f^p : \wp(X)^n \to \wp(Y)$ to powersets is defined as $f^p(S_1, ..., S_n) \stackrel{\text{def}}{=} \{f(x_1, ..., x_n) \mid 1 \leq i \leq n, x_i \in S_i\}$.

2.1.2 Ordered structures

It is useful to work with structures that, unlike sets, embody the relations existing between their elements. Let us first consider structures obtained by combining sets and their ordering relations.

Definition 2.5. A set C with ordering relation $\leq$ is a partial ordered set, also called *poset*, and it is denoted as $\langle C, \leq \rangle$.

Let us consider two elements x and y of a poset $\langle C, \leq \rangle$. We say that x is *covered by* y in C, written $x \prec y$, if $x < y$ and there is no $z \in C$ with $x < z < y$. Relation $\prec$ can be used to define a *Hasse diagram* for a finite ordered set C: the elements of C are represented by points in the plane, where x is drown above y if $x < y$, and a line is drown from point x to point y precisely when $x \prec y$. The following figure shows the graphical representation of the ordered sets $C1 = \{a, b, c, d, f, g\}$ and $C2 = \{a, b, c, d, e, g\}$ in which $a < b, b < d, b < e, d < f, d < g, c < e, e < g$.

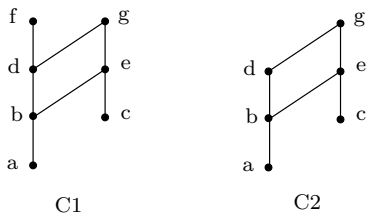


Fig. 2.1. Hasse diagram of $C1$ and $C2$

In particular, given a poset $\langle C, \leq \rangle$, if all pairs of elements of C are in ordering relation $\leq$, then $\leq$ is a total order and C is a *chain*.

Definition 2.6. Let $\langle C, \leq \rangle$ be a poset. C is a *chain* if $\forall x, y \in C : x \leq y$ or $y \leq x$.

Hence, a chain is a totally ordered set. A typical example of a partial order set is the powerset $\wp(X)$ of any set X, ordered by subset inclusion. This is a partial order, in fact given $x, y, z \in X$ we have that $\{x, y\} \not\subseteq \{y, z\}$ and $\{y, z\} \not\subseteq \{x, y\}$. On the other side, the set of numbers with the standard ordering relation is a typical example of a chain. Given a poset $\langle C, \leq \rangle$ we denote with $\langle C^\delta, \leq_\delta \rangle$ its *dual*, where $x \leq_\delta y$ if and only if $y \leq x$. This definition leads to the following principle.

Definition 2.7. Given any statement Φ true on all posets, its dual Φ^δ holds for all posets.

Given a poset $\langle C, \leq \rangle$ it is possible to define two interesting families of sets of elements in C based on the ordering relation $\leq$.

Definition 2.8. Let $\langle C, \leq \rangle$ be a poset. A subset $Q \subseteq C$ is an *ideal* of C, if we have that $\forall x \in Q, y \in C : y \leq x \Rightarrow y \in Q$. A subset $Q \subseteq C$ is a *filter* if it is the dual of an ideal.

Observe that these sets can be built starting from a general subset of C. The *filter closure* (or upward closure) of a set $Q \subseteq C$, is given by $\uparrow Q \stackrel{\text{def}}{=} \{y \in C \mid \exists x \in Q : x \leq y\}$, where $\langle C, \leq \rangle$ is a poset. The *ideal closure* (or downward closure) $\downarrow Q$ is dually defined. In the following we use the shortland $\downarrow x$ (resp. $\uparrow x$) for $\downarrow \{x\}$ (resp. $\uparrow \{x\}$). For example, considering the poset $C1$ in Fig. 2.1, we have that the sets $\{c\}$, $\{a, b, c, d, e\}$ and $\{a, b, d, f\}$ are all ideals, while $\{b, d, e\}$ is not an ideal and $\downarrow \{b, d, e\} = \{a, b, c, d, e\}$. Moreover, the set $\{e, f, g\}$ is a filter, while $\{a, b, d, f\}$ is not a filter and $\uparrow \{a, b, d, f\} = \{a, b, d, e, f, g\}$.

Definition 2.9. Let $\langle C, \leq \rangle$ be a poset, and let $X \subseteq C$. An element a is an *upper bound* of X if $\forall x \in X : x \leq a$, if a belongs also to X it is the *maximal*. The smallest element of the set of upper bounds of X, when it exists, is called the *least upper bound* (*lub* or *sup* or *join*) of X, and it is denoted as $\bigvee X$. When the *lub* belongs to C it is called *maximum* (or *top*) and it is usually denoted as $\top$.

Considering the ordered sets in Fig. 2.1 we have that: the set $\{a, b, c\}$ has least upper bound e, $C1$ has maximal elements f and g and no greatest element, while $C2$ has greatest element g. The notions of *lower bound*, *minimal element*, *greatest lower bound* (*glb* or *inf* or *meet*) of a set X, denoted $\bigwedge X$, and *minimum* (or *bottom*), denoted by $\bot$ are dually defined. It is clear that, if a poset has a top (or bottom) element from the antisymmetry property of the ordering relation, it is unique. In the following we use $x \wedge y$ and $x \vee y$ to denote respectively the elements $\bigwedge\{x, y\}$ and $\bigvee\{x, y\}$.

Algebraic ordered structures can be further characterized. A poset C is a *direct set* if each non-empty finite subset of C has least upper bound in C. A typical example of a direct set is a chain.

Definition 2.10. A *complete partial order* (or *cpo*) is a poset $\langle C, \leq \rangle$ such that $\bot \in C$ and for each direct set D in C we have that $\bigvee D \in C$.

It is clear that every finite poset is a cpo. Moreover, it holds that a poset C is a cpo if and only if each chain in C has least upper bound.

Definition 2.11. A poset $\langle C, \leq \rangle$, with $C \neq \varnothing$, is a *lattice* if $\forall x, y \in C$ we have that $x \vee y$ and $x \wedge y$ belong to C. A lattice is *complete* if for every $S \subseteq C$ we have that $\bigvee S \in C$ and $\bigwedge S \in C$.

As usual, a complete lattice C with ordering relation $\leq$, lub $\vee$, glb $\wedge$, top element $\top = \bigvee C = \bigwedge \varnothing$, bottom element $\bot = \bigwedge C = \bigvee \varnothing$, is denoted as $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$. Often, $\leq_C$ will be used to denote the underlying ordering of poset C, and $\vee_C$, $\wedge_C$, $\top_C$ and $\bot_C$ denote the basic operations and elements of

a complete lattice C. Observe that the ordered sets in Fig. 2.1 are not lattices since elements a and c do not have glb. The set $\mathbb{N}$ of natural numbers with the standard ordering relation is a lattice where the glb and the lub of a set are given respectively by its minimum and maximum element. However, $\langle \mathbb{N}, \leq \rangle$ is not complete because any infinite subset of $\mathbb{N}$, as for example $\{n \in \mathbb{N} \mid n > 100\}$, has no lub. On the other hand, an example of complete lattice often used in the thesis, is the powerset $\wp(X)$, where X is any set. In this case the ordering is given by set inclusion, the glb by the intersection of sets and the lub by the union of sets.

In the following we use the term *domain* to refer to a generic ordered structure. Let us introduce the notion of *Moore family*, which is a particular complete lattice that plays a crucial role in abstract interpretation.

Definition 2.12. Let C be a complete lattice. The subset $X \subseteq C$ is a *Moore family* of C if $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\bigwedge S \mid S \subseteq X\}$, where $\bigwedge \varnothing = \top \in \mathcal{M}(X)$.

This particular lattice can be built starting form a subset $X \subseteq C$ through the *Moore closure* (or *meet closure*) $\mathcal{M}$. In fact $\mathcal{M}(X)$ is the smallest, with respect to set inclusion, subset of C containing X and being a Moore family of C. In a lattice it is possible to characterize some particular elements called *meet-irreducible* (resp. *join-irreducible*) based on the meet (resp. join) operator.

Definition 2.13. Let C be a lattice. An element e of C such that $e \neq \top$ is *meet-irreducible* if $e = x \wedge y$ implies that $e = x$ or $e = y$. Let $Mirr(C)$ denote the set of meet-irreducible elements of C.

A lattice C is *meet-generated* by its meet-irreducibles if the Moore closure of its meet-irreducible elements generates each element of C, i.e., $C = \mathcal{M}(Mirr(C))$. The notions of *join-irreducible* elements and *join-generated* lattice are dually defined.

Definition 2.14. A poset C satisfies the *ascending chain condition (ACC)* if for each $x_1 \leq x_2 \leq ... \leq x_n \leq ...$ increasing sequence of elements of C, there exists k such that: $x_k = x_{k+1} = ...$

It is clear that the ordered set of even numbers $\{n \in \mathbb{N} \mid n \bmod 2 = 0\}$ does not satisfy the ascending chain condition, since the ascending chain of even numbers does not converge. A poset satisfying the *descending chain condition (DCC)* is dually defined as a poset without infinite descending chains.

An interesting operation on the elements of a complete lattice is the *complement*.

Definition 2.15. Let C be a poset with $\bot$ and $\top$. Given an element $x \in C$ we say that $y \in C$ is the *complement* of x if $x \wedge y = \bot$ and $x \vee y = \top$.

A *complemented* lattice is a lattice where each element has a complement.

Definition 2.16. A complemented and distributive lattice is called a *Boolean algebra*, where distributive means that for each $x, y, z \in C$: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.

A *complete boolean algebra* is a complete lattice which is both complemented and distributive. There is another possible notion of complementation known as pseudo-complement.

Definition 2.17. Consider an element x of a lattice C. An element x^* is a *pseudo-complement* of x if: $x \wedge x^* = \bot$ and $\forall y \in C : x \wedge y = \bot \Rightarrow y \leq x^*$.

Observe that if a pseudo-complement exists then it is unique, so that we can refer to the pseudo-complement of a given element. Thus, the pseudo-complement of an element x is the greatest element, whose glb with x returns bottom, while it has no condition on the lub. A *pseudo-complemented lattice* is a lattice where each element has a pseudo-complement. Considering the lattice in Fig. 2.2 we have that the complement of element a is d and viceversa, while we have the following pseudo-complements: $a^* = d$, $c^* = d$ and $d^* = a$.

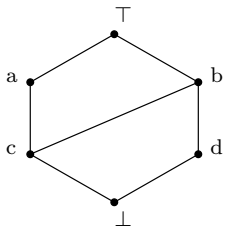


Fig. 2.2. Pseudo-complement

Given a lattice C, the *relative pseudo-complement* of a pair of elements $x, y \in C$ is $x * y : x \wedge x * y \leq y$ and, for each $z \in C$ we have that if $x \wedge z \leq y$ then $x \leq x * y$. A lattice C is *relatively pseudo-complemented* if the relative pseudo-complement of each $x, y \in C$ belongs to C.

Functions on domains

Let us consider the functions on domains and their properties.

Definition 2.18. Let $\langle C, \leq_C \rangle$ and $\langle D, \leq_D \rangle$ be two posets, and consider a function $f : C \to D$, then:

- f is *monotone* (or *order preserving*) if for each $x, y \in C$ such that $x \leq_C y$ we have that $f(x) \leq_D f(y)$;
- f is *order embedding* if for every $x, y \in C$ we have that $x \leq_C y \Leftrightarrow f(x) \leq_D f(y)$;

– f is an *order isomorphism* if f is order embedding and surjective.

The continuous and additive functions are particularly important when studying program semantics.

Definition 2.19. Given two cpo C and E, a function $f : C \to E$ is (*Scott*)-*continuous* if it is monotone and if it preserves the limits of direct sets, namely if for each direct set D of C, we have $f(\bigvee_C D) = \bigvee_E f(D)$.

Co-continuous functions can be defined dually.

Definition 2.20. Given two cpo C and D, a function $f : C \to D$ is (*completely*) *additive* if for each subset $X \subseteq C$, we have that $f(\bigvee_C X) = \bigvee_D f(X)$.

Hence, an additive function preserves the limits (lub) of all subsets of C (empty-set included), meaning that an additive function is also continuous. The notion of *co-additive* functions is dually defined.

We use the symbol $\sqsubseteq$ to denote the pointwise ordering between functions: if X is any set, C is a poset and $f, g : X \to C$ then $f \sqsubseteq g$ if for all $x \in X$: $f(x) \leq_C g(x)$.

2.1.3 Fixpoints

Definition 2.21. Let $f : C \to C$ be a function on a poset C. An element $x \in C$ is a *fixpoint* of f if $f(x) = x$. Let $\mathrm{Fix}(f) \stackrel{\text{def}}{=} \{x \in C \mid f(x) = x\}$ be the set of all fixpoints of function f.

Thanks to the ordering relation $\leq_C$ on C, we can define the least fixpoint of f, denoted $\mathit{lfp}^{\leq_C}(f)$ (or simply $\mathit{lfp}(f)$ when the ordering relation is clear from the context), as the unique element $x \in \mathrm{Fix}(f)$ such that for all $y \in \mathrm{Fix}(f) : x \leq_C y$. The notion of greatest fixpoint, denoted $\mathit{gfp}^{\leq_C}(f)$ (or simply $\mathit{gfp}(f)$ when the ordering relation is clear from the context), is dually defined. Let us recall the well known Knaster-Tarski's fixpoint theorem.

Theorem 2.22. Given a complete lattice $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ and a monotone function $f : C \to C$, then the set of fixpoints of f is a complete lattice with ordering $\leq$. In particular, if f is continuous, the least fixpoint can be characterized as:

$$\mathit{lfp}(f) = \bigvee_{n \leq \omega} f^n(\bot)$$

where, given $x \in C$, the i-th power of f in x is inductively defined as follows $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$.

Hence, the least fixpoint of a continuous function on a complete lattice can be computed as the limit of the iteration sequence obtained starting from the bottom of the lattice. Dually, the greatest fixpoint of a co-continuous function f on a complete lattice C, can be computed staring from the top of the lattice, namely $\mathit{gfp}(f) = \bigwedge_{n \leq \omega} f^n(\top)$.

2.1.4 Closure operators

Let us introduce the notion of *closure operator*, which is very important when dealing, for example, with abstract interpretation.

Definition 2.23. An *upper closure operator*, or simply a *closure*, on a poset $\langle C, \leq \rangle$ is an operator $\rho : C \to C$ that is:

- extensive: $\forall x \in C : x \leq \rho(x)$;
- monotone: $\forall x, y \in C : x \leq y \Rightarrow \rho(x) \leq \rho(y)$;
- idempotent: $\forall x \in C : \rho(\rho(x)) = \rho(x)$.

Function $f : C \to C$ in Fig. 2.3 (a) is an upper closure operator while function $g : C \to C$ in Fig. 2.3 (b) is not since it is not idempotent.

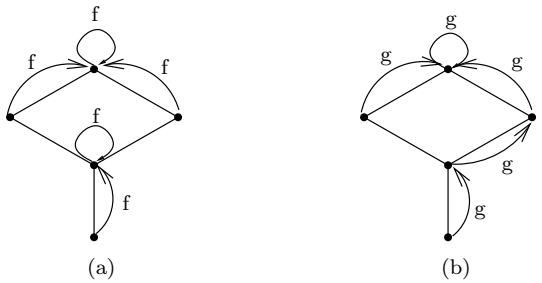


Fig. 2.3. f is an upper closure operator while g is not

Let $uco(C)$ denote the set of all upper closures operators of domain C. If $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ is a complete lattice, then for each closure operator $\rho \in uco(C)$ we have that:

$$\rho(c) = \bigwedge \{\ x \in C \,|\, x = \rho(x), c \leq x\ \}$$

meaning that the image of an elements c through ρ is the minimum fixpoint of ρ greater than c. Moreover, ρ is uniquely determined by its image $\rho(C)$, that is the set of its fixpoints $\rho(C) = \mathrm{Fix}(\rho)$. In fact, the following properties of a closure operator have been proved [149]:

- if $\rho \in uco(C)$ then $\rho(C) \subseteq C$ is a Moore family;
- if $X \subseteq C$ is a Moore family then $\eta_X : C \to C$ is a closure on C, where $\lambda c.\eta(c) = \bigwedge \{\ x \in X \,|\, c \leq x\ \}$;
- moreover, it holds that: $\eta_X(C) = X$ and $\eta_{\rho(C)} = \rho$.

In the following, the notation ρ denotes closures defined both as functions or Moore families. Observe that, given a complete lattice C, the Moore closure operator $\mathcal{M} : \wp(C) \to \wp(C)$, i.e., $\mathcal{M}(X) = \{\bigwedge Y \mid Y \subseteq X\}$, is a closure on

the powerset $\wp(C)$ ordered by set inclusion. Thus, given $X \subseteq C$, we have that $\mathcal{M}(X)$ can be characterized as the smallest set meet-closed in C that contains X.

Given a complete lattice C and a closure $\rho \in uco(C)$, the image $\rho(C)$ is a complete lattice $\langle \rho(C), \leq_C, \vee_\rho, \wedge_C, \top_C, \rho(\bot_C)\rangle$ on the ordering $\leq_C$ inherited from C, where:

- the lub is defined as $\vee_\rho(X) = \rho(\vee_C X)$ for every $X \subseteq \rho(C)$;
- the glb and the top element coincides with the ones of C;
- the bottom element is given by the image of the bottom of C, i.e., $\rho(\bot_C)$.

Given the closures $\rho, \eta \in uco(C)$ and a subset $Y \subseteq C$ we have that:

- $\rho(\bigwedge \rho(Y)) = \bigwedge \rho(Y)$;
- $\rho(\bigvee Y) = \rho(\bigvee \rho(Y))$;
- $\eta \sqsubseteq \rho \Leftrightarrow \eta \circ \rho = \rho \Leftrightarrow \rho \circ \eta = \rho$;
- $\rho \circ \eta \in uco(C) \Leftrightarrow \rho \circ \eta = \eta \circ \rho = \eta \sqcup \rho$.

An important result on closures states that the set of closure of a complete lattice is a complete lattice with respect to the pointwise ordering on functions. In particular, given a complete lattice C, then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x\rangle$ is a complete lattice [149], where for each $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i\in I} \subseteq uco(C)$ and $x \in C$ we have that:

- $\rho \sqsubseteq \eta$ iff $\forall c \in C : \rho(c) \leq \eta(c)$ iff $\eta(C) \subseteq \rho(C)$;
- $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$;
- $(\sqcup_{i\in I}\rho_i)(x) = x \Leftrightarrow \forall i \in I : \rho_i(x) = x$;
- $\lambda x.\top$ is the top element and $\lambda x.x$ is the bottom element.

2.1.5 Galois connections

Another notion typically used in abstract interpretation is the one of Galois connection.

Definition 2.24. Tow posets $\langle C, \leq_C\rangle$ and $\langle D, \leq_D\rangle$ and two monotone functions $\alpha : C \to D$ and $\gamma : D \to C$ such that:

- $\forall c \in C : c \leq_C \gamma(\alpha(c))$ and
- $\forall d \in D : \alpha(\gamma(d)) \leq_D d$,

form a *Galois connection*, equivalently denoted by (C, α, γ, D) or $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D$.

The definition of Galois connection is equivalent to the one of *adjunction* between C and D, where (C, α, γ, D) is an adjunction if:

$$\forall c \in C, \forall d \in D : \alpha(c) \leq_D d \Leftrightarrow c \leq_C \gamma(d)$$

In this case α (resp. γ) is called the *right adjoint* (resp. *left adjoint*) of γ (resp. α).

A Galois connection (C, α, γ, D) where $\forall d \in D : \alpha(\gamma(d)) = d$ is called a *Galois insertion*. A Galois insertion is denoted also by $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D$. Observe that a Galois connection (C, α, γ, D) can be *reduced* to a Galois insertion collecting together all the elements $d \in D$ that have the same image under γ.

There are a number of interesting properties that hold on Galois connections. In particular, if (C, α, γ, D) and (C, α', γ', D) are two Galois connections then $\alpha = \alpha' \Leftrightarrow \gamma = \gamma'$. In fact, it is possible to prove that given a Galois connection (C, α, γ, D) each function can be uniquely determined by the other one, in fact given $c \in C$ and $d \in D$ we have that:

- $\alpha(c) = \bigwedge_D \{\ y \in D \,|\, c \leq_C \gamma(y)\ \}$;
- $\gamma(d) = \bigvee_C \{\ x \in C \,|\, \alpha(x) \leq_D \in d\ \}$.

Thus, in order to specify a Galois connection it is enough to provide the right or left adjoint since the other one is uniquely determined by the above equalities. Moreover, it has been proved that given a Galois connection (C, α, γ, D) the function α preserves existing lub (i.e., if $X \subseteq C$ and $\exists \bigvee_C X \in C$ then $\exists \bigvee_D \alpha(X) \in D$ and $\alpha(\bigvee_C X) = \bigvee_D \alpha(X)$) and γ preserves existing glb. In particular, when C and D are complete lattices we have that α is additive and γ is co-additive. Thus, given two complete lattices C and D, each additive function $\alpha : C \to D$ or co-additive function $\gamma : D \to C$ determines a Galois connection (C, α, γ, D) where:

- $\forall y \in D : \gamma(y) = \bigvee_C \{\ x \in C \,|\, \alpha(x) \leq_D y\ \}$;
- $\forall x \in C : \alpha(x) = \bigwedge_D \{\ y \in D \,|\, x \leq_C \gamma(y)\ \}$.

This means that, α maps each element $c \in C$ in the smallest element in D whose image under γ is greater than c as regards $\leq_C$. Viceversa, γ maps each element $d \in D$ in the greatest element in C whose image by α is lower than d as regards $\leq_D$. Given a Galois connection (C, α, γ, D) where C and D are posets we have that:

- if C has a bottom element $\bot_C$, then D has bottom element $\alpha(\bot_C)$;
- dually, if D has top element $\top_D$, then C has top element $\gamma(\top_D)$;
- $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$;
- if (D, α', γ', E) is a Galois connection, then $(C, \alpha' \circ \alpha, \gamma \circ \gamma', E)$ is a Galois connection, namely it is possible to compose Galois connections;
- if (C, α, γ, D) is a Galois insertion and C is a complete lattice, then D is a complete lattice;
- α is surjective if and only if γ is injective if and only if (C, α, γ, D) is a Galois insertion.

From the last property above we have that a Galois insertion between two complete lattices C and D is fully specified by a surjective and additive map $\alpha : C \to D$ or by an injective and co-additive map $\gamma : D \to C$.

Two Galois connections $(C_1, \alpha_1, \gamma_1, D_1)$ and $(C_2, \alpha_2, \gamma_2, D_2)$ are isomorphic, denoted $\cong$, if $C_1 \cong C_2$, $D_1 \cong D_2$ and functions α_1, α_2 and γ_1, γ_2 coincide up to isomorphism. It is possible to show that this holds if and only if $\gamma_1(D_1) \cong \gamma_2(D_2)$. In particular, when $C_1 = C_2$, this holds if and only if $\gamma_1(D_1) = \gamma_2(D_2)$.

2.1.6 Galois connections and closure operators

The notion of Galois connection and the one of closure operators are closely related. Given a Galois connection (C, α, γ, D) we can prove that the map $\gamma \circ \alpha$ is an upper closure on C, i.e., $\gamma \circ \alpha \in uco(C)$. Moreover, if C is a complete lattice then $\gamma(D)$ is a Moore family of C. On the other side, given a poset C and a closure $\rho \in uco(C)$ then $(C, \rho, \lambda x.x, \rho(C))$ defines a Galois insertion. Moreover, we have that:

- if (C, α, γ, D) is a Galois insertion then $(C, \gamma \circ \alpha, \lambda x.x, \gamma(\alpha(C))) \cong (C, \alpha, \gamma, D)$;
- the closure on C defined by the Galois insertion $(C, \rho, \lambda x.x, \rho(C))$ induced by the closure $\rho \in uco(C)$ trivially coincides with ρ.

Thus, the notions of Galois insertion and closure operators are equivalent. This holds also for Galois connections up to reduction.

2.2 Abstract Interpretation

According to a widely recognized definition: *"Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems"* [39]. The key idea of abstract interpretation is that the behaviour of a program at different levels of abstraction is an approximation of its (concrete) semantics. Let $\mathbf{S}$ denote a formal definition of the semantics of programs in $\mathbb{P}$ written in a certain programming language, and let C be the semantic domain on which $\mathbf{S}$ is computed. Let us denote with $\mathbf{S}^\sharp$ an *abstract semantics* expressing an approximation of the *concrete semantics* $\mathbf{S}$. The definition of the abstract semantics $\mathbf{S}^\sharp$ is given by the definition of the concrete semantics $\mathbf{S}$ where the domain C has been replaced by an approximated semantic domain A in Galois connection with C, i.e., (C, α, γ, A). Then, the abstract semantics is obtained by replacing any function $F : C \to C$, used to compute $\mathbf{S}$, with an approximated function $F^\sharp : A \to A$ that correctly mimics the behaviour of F in the domain properties expressed by A.

Concrete and Abstract Domains

The concrete program semantics $\mathbf{S}$ of a program $P \in \mathbb{P}$ is computed on the so-called *concrete domain*, i.e., the poset of mathematical objects on which the program runs, here denoted by $\langle C, \leq_C \rangle$. The ordering relation encodes relative precision: $c_1 \leq_C c_2$ means that c_1 is a more precise (concrete) description than c_2. For instance, the concrete domain for a program with integer variables is simply given by the powerset of integer numbers ordered by subset inclusion $\langle \wp(\mathbb{Z}), \subseteq \rangle$.

Approximation is encoded by an *abstract domain* $\langle A, \leq_A \rangle$, which is a poset of abstract values that represent some approximated properties of concrete objects. Also in the abstract domain, the ordering relation models relative precision: $a_1 \leq_A a_2$ means that a_1 is a better approximation (i.e., more precise) than a_2. For example, we may be interested to the sign of an integer variable, so that a simple abstract domain for this property may be $\mathit{Sign} = \{\top, 0-, 0, 0+, \bot\}$ where $\top$ gives no sign information, $0-/0/0+$ state that the integer variable is negative/zero/positive, while $\bot$ represent an uninitialized variable or an error for a variable (e.g., division by zero): thus, we have that $\bot < 0 < 0- < \top$ and $\bot < 0 < 0+ < \top$, so that, in particular, the abstract values $0-$ and $0+$ are incomparable.

As observed earlier, in standard abstract interpretation, concrete and abstract domains are related through a Galois connection (C, α, γ, A). In this case, $\alpha : C \to A$ is called the *abstraction function* and $\gamma : A \to C$ the *concretization function*. Given a Galois connection (C, α, γ, A), we say that A is an *abstraction* (or *abstract interpretation*) of C, and that C is a *concretization* of A. The abstraction and concretization maps express the meaning of the abstraction process: $\alpha(c)$ is the abstract representation of c, and $\gamma(a)$ represents the concrete meaning of a. Thus, $\alpha(c) \leq_A a$ and, equivalently, $c \leq_C \gamma(a)$ means that a is a sound approximation in A of c. Galois connections, being adjunctions, ensure that $\alpha(c)$ actually provides the best possible approximation in the abstract domain A of the concrete value $c \in C$. In the abstract domain *Sign*, for example, we have that $\alpha(\{-1, -5\}) = 0-$ while $\alpha(\{-1, +1\}) = \top$. This confirms the fact that Galois connection is the right tool for modeling the approximation process. Moreover, closure operators $\rho \in \mathit{uco}(C)$, being equivalent to Galois connections, have properties (monotonicity, extensivity and idempotency) that well fit the abstraction process. The monotonicity ensures that the approximation process preserves the relation of being more precise than. If a concrete element c_1 contains more information than a concrete element c_2, then after approximation we have that $\rho(c_1)$ is more precise than $\rho(c_2)$. Approximating an object means that we could loose some of its properties, therefore it is not possible to gain any information during approximation. Hence, when approximating an element we obtain an object that contains at most the same amount of information of the

original object. This is well expressed by the fact that the closure operator is extensive. Finally, we have that the approximation process looses information only on its first application, namely if the approximated version of the object c is the element a, then approximating a we obtain a. Meaning that the approximation function as to be idempotent. Hence, it is possible to describe abstract domains on C in terms of both Galois connections and upper closure operators [43]. The formulation of abstract domains through upper closures is particularly convenient when reasoning about properties of abstract domains independently from the representation of their objects (i.e., independently from the names of objects in A).

Of course, abstract domains can be compared with respect to their relative degree of precision: if A_1 and A_2 are both abstract domains of a common concrete domain C, we have that A_1 is more precise than A_2, denoted by $A_1 \sqsubseteq A_2$, when for any $a_2 \in A_2$ there exists $a_1 \in A_1$ such that $\gamma_1(a_1) = \gamma_2(a_2)$, i.e., when $\gamma_2(A_2) \subseteq \gamma_1(A_1)$. This ordering relation on the set of all possible abstract domains defines the *lattice of abstract interpretations*.

Consider the concrete domain given by the powerset of integers $\langle \wp(\mathbb{Z}), \subseteq \rangle$, and assume that we are interested in the sign of a given integer number. Fig. 2.4 presents some possible abstractions of $\wp(\mathbb{Z})$ expressing properties on the sign of integers. The abstraction and concretization functions are the obvious ones

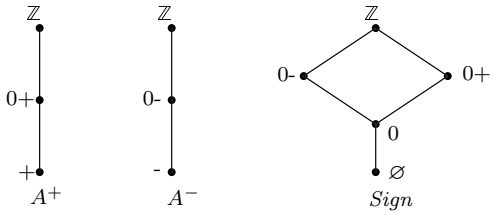


Fig. 2.4. Abstractions of $\wp(\mathbb{Z})$

(e.g., $\alpha(\{0, -1, -2\}) = 0-$, $\alpha(\{-1, 2\}) = \mathbb{Z}$, while $\gamma(0+) = \{z \geq 0\}$ and $\gamma(-) = \{z < 0\}$). It is easy to show that A^+, A^- and *Sign* are in Galois connection with $\wp(\mathbb{Z})$ and that the abstract domain *Sign* is more abstract, i.e., less precise, than both A^+ and A^-, while A^+ and A^- are incomparable. The examples provided in the rest of the chapter will often refer to the abstract domain of *Sign* thanks to its simplicity.

The abstract domain of intervals

When considering the concrete domain of the powerset of integers a non trivial and well known abstraction is given by the abstract domain of intervals, here

denoted by $\langle Interval, \leq_I \rangle$ [117]. The elements of the *Interval* domain are defined by the following:

$$Interval \stackrel{\text{def}}{=} \{\bot\} \cup \{[l,h] \mid l \leq h, l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}\}$$

where the standard ordering on integers is extended to $\mathbb{Z} \cup \{+\infty, -\infty\}$, by setting that $-\infty \leq +\infty$ and that for all $z \in \mathbb{Z}$: $z \leq +\infty$ and $-\infty \leq z$. The idea is that the abstract element $[l,h]$ corresponds to the interval from l to h including the end points if they are in $\mathbb{Z}$, while $\bot$ denotes the empty interval. Intuitively an interval int_1 is smaller than an interval int_2, denoted $int_1 \leq_I int_2$, when int_1 is contained in int_2. Formally we have:

- for all $int \in Interval$: $\bot \leq_I int \leq (-\infty, +\infty)$;
- for all $l_1, l_2 \in \mathbb{Z} \cup \{-\infty\}, h_1, h_2 \in \mathbb{Z} \cup \{+\infty\}$: $[l_1,h_1] \leq_I [l_2,h_2] \Leftrightarrow l_1 \geq l_2 \wedge h_1 \leq h_2$;

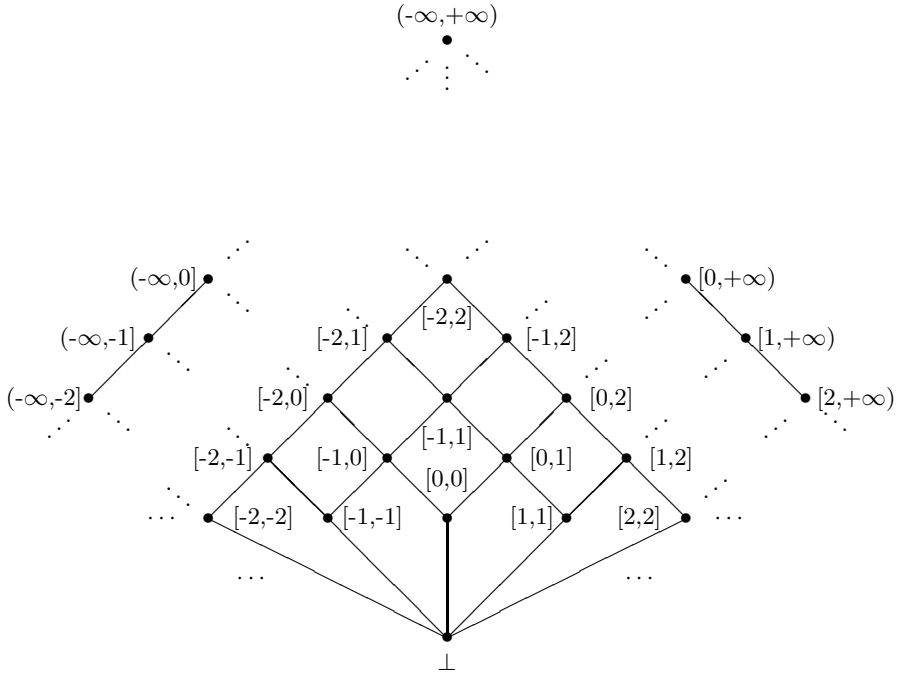


Fig. 2.5. The Interval abstract domain.

Fig. 2.5 represents the abstract domain of intervals. $(\wp(\mathbb{Z}), \alpha_I, \gamma_I, Interval)$ is a Galois insertion where the abstraction $\alpha_I : \wp(\mathbb{Z}) \to Interval$ and concretization $\gamma_I : Interval \to \wp(\mathbb{Z})$ maps are defined as follows, let $l, h \in \mathbb{Z}$ then:

$$\alpha_I(S) = \begin{cases} \bot & \text{if } S = \varnothing \\ [l,h] & \text{if } min(S) = l \wedge max(S) = h \\ (-\infty,h] & \text{if } \nexists min(S) \wedge max(S) = h \\ [l,+\infty) & \text{if } min(S) = l \wedge \nexists max(S) \\ (-\infty,+\infty) & \text{if } \nexists min(S) \wedge \nexists max(S) \end{cases}$$

$$\gamma_I(int) = \begin{cases} \varnothing & \text{if } int = \bot \\ \{z \in \mathbb{Z} \mid l \leq z \leq h\} & \text{if } int = [l,h] \\ \{z \in \mathbb{Z} \mid z \leq h\} & \text{if } int = (-\infty,h] \\ \{z \in \mathbb{Z} \mid z \geq l\} & \text{if } int = [l,+\infty) \\ \mathbb{Z} & \text{if } \mathbb{int} = (-\infty,+\infty) \end{cases}$$

For example, the set $\{2,5,8\}$ is abstracted in the interval $[2,8]$, while the infinite set $\{z \in \mathbb{Z} \mid z \geq 10\}$ is abstracted in the interval $[10,+\infty)$. It is possible to prove that $\langle Interval, \leq_I \rangle$ is a complete lattice [41] with top element given by $(-\infty,+\infty)$, bottom element given by $\bot$, glb $\sqcap_I$ and lub $\sqcap_I$ defined in the following.

$$[l_1,h_1] \sqcap_I [l_2,h_2] = [max(\{l_1,l_2\}), min(\{h_1,h_2\})]$$

For example, $[2,10] \sqcap_I [5,20] = [5,10]$ and $[2,10] \sqcap (-\infty,5] = [2,5]$, while $[2,10] \sqcap_I [20,25] = \bot$. Thus, the glb of a set of intervals returns the bigger interval contained in all of them.

$$[l_1,h_1] \sqcup_I [l_2,h_2] = [min(\{l_1,l_2\}), max(\{h_1,h_2\})]$$

For example, $[2,10] \sqcup_I [5,20] = [2,20]$ and $[2,10] \sqcup_I (-\infty,5] = (-\infty,10]$, while $[2,10] \sqcup_I [20,25] = [2,25]$. Hence, the lub of a set of intervals returns the smallest interval that contains all of them.

It is clear that the abstract domain of intervals and the abstract domain of sign can be compared with respect to their degree of precision. In particular, *Interval* provides a more precise representation of the powerset of integers than what *Sign* does, meaning that $Interval \sqsubseteq Sign$.

2.2.1 Lattice of abstract interpretations

The ordering relation between abstract domains corresponds precisely to the pointwise ordering of the corresponding closure operators on $uco(C)$. In particular, consider two Galois connections $(C, \alpha_1, \gamma_1, A_1)$ and $(C, \alpha_2, \gamma_2, A_2)$ and the corresponding closure operators $\rho_1, \rho_2 \in uco(C)$, i.e., $\rho_i(C) \cong A_i$, then A_1 is more precise than A_2, i.e., $A_1 \sqsubseteq A_2$, iff $\rho_1 \sqsubseteq \rho_2$ in $uco(C)$ iff $\rho_2(C) \subseteq \rho_1(C)$. Thus, given a domain C, $\langle uco(C), \sqsubseteq \rangle$ is isomorphic to the lattice of abstract interpretations introduced earlier. This is the reason why the symbol $\sqsubseteq$ is used also to compare abstract domains with respect to their relative precision. Let us see the meaning of least upper bound and greatest lower bound as operators on domains.

Least common abstraction

The lub operator $\sqcup$ on $uco(C)$ corresponds to the computation of the least common abstraction. In particular, consider the set $\{A_i\}_{i\in I} \subseteq uco(C)$ of abstraction, then $\sqcup_{i\in I} A_i$ is the *least* (with respect to $\sqsubseteq$) *common abstraction* of all the A_i's, i.e., the most concrete domain in $uco(C)$ which is abstraction of all A_i's. In particular, $(\sqcup_{i\in I}\rho_i(C)) = \bigcap_{i\in I}\rho_i(C)$. In Fig. 2.6 we consider two abstractions of $\wp(\mathbb{Z})$, $Sign^+$ and $Parity_0$ expressing respectively the sign and parity of integer numbers (*ev* represents all the even integers and *od* all the odd integers), and the domain obtained by their intersection, expressing their least common abstraction.

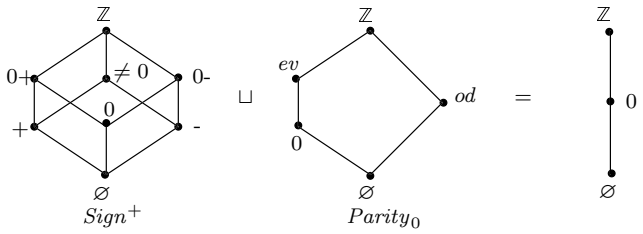


Fig. 2.6. Least upper bound of closures

Reduced Product

On the other side, the glb operator $\sqcap$ on $uco(C)$ is called the *reduced product* (basically cartesian product plus reduction) [37,43]. In particular, $\sqcap_{i\in I} A_i$ is the most abstract domain in $uco(C)$, which is more concrete than every A_i's. Let us remark that $\sqcap_{i\in I} A_i = \mathcal{M}(\bigcup_{i\in I} A_i)$. The reduced product is typically used to combine known abstract domains in order to design new abstractions. In Fig. 2.7 we consider the domain *Sign* and *Parity*, abstractions of $\wp(\mathbb{Z})$, and their reduced product.

Pseudo-complement

Complementation (or pseudo-complement) corresponds to the inverse of reduced product [37, 57], namely an operator that, given two domains $C \sqsubseteq D$, gives as result the most abstract domain $C \ominus D$, whose reduced product with D is exactly C, i.e., $(C \ominus D) \sqcap D = C$. Because of the peculiar structure of abstract domains in abstract interpretation, the pseudo-complement of an abstract domain does not correspond to the set theoretic complement $C \setminus D$. This because the result would not be in general an abstract domain. Thus, the pseudo-complement of an abstract domain D is defined as:

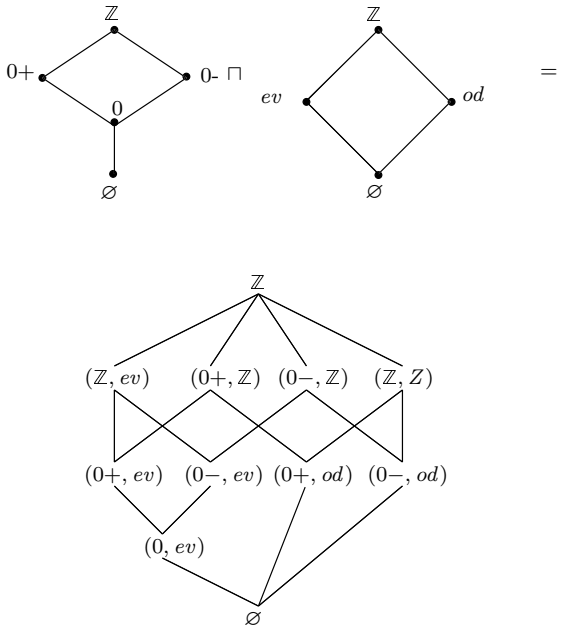


Fig. 2.7. Reduced product of closures

$$C \ominus D \stackrel{\text{def}}{=} \sqcup \{E \in uco(C) | D \sqcap E = C\}$$

Fig. 2.8 considers the *Sign* domain and one of its abstraction and computes the complement domain.

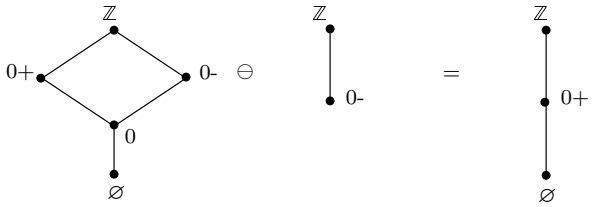


Fig. 2.8. Abstract domain complementation

2.2.2 Abstract Operations

Soundness

In abstract interpretation, a concrete semantic operation is formalized as any (possibly n-ary) function $f : C \to C$ on the concrete domain. For example,

a (unary) integer squaring operation sq on the concrete domain $\wp(\mathbb{Z})$ is given by $sq(X) = \{x^2 \in \mathbb{Z} \mid x \in X\}$, while an integer increment (by one) operation $plus$ is given by $plus(X) = \{x + 1 \in \mathbb{Z} \mid x \in X\}$. A concrete semantic operation must be approximated on some abstract domain A by a *sound abstract operation* $f^\sharp : A \to A$. This means that $f^\sharp$ must be a correct approximation of f in A: for any $c \in C$ and $a \in A$, if a approximates c then $f^\sharp(a)$ must approximate $f(c)$. This is therefore encoded by the condition:

$$\forall c \in C : \alpha(f(c)) \leq_A f^\sharp(\alpha(c)) \tag{2.1}$$

For example, a correct approximation $sq^\sharp$ of sq on the abstract domain *Sign* can be defined as follows: $sq^\sharp(\bot) = \bot$, $sq^\sharp(0) = 0$, $sq^\sharp(0-) = 0+$, $sq^\sharp(0+) = 0+$ and $sq^\sharp(\top) = \top$; while a correct approximation $plus^\sharp$ of $plus$ on *Sign* is given by: $plus^\sharp(\bot) = \bot$, $plus^\sharp(0-) = \top$, $plus^\sharp(0) = 0+$, $plus^\sharp(0+) = 0+$ and $plus^\sharp(\top) = \top$. Soundness can be also equivalently stated in terms of the concretization map:

$$\forall a \in A : f(\gamma(a)) \leq_C \gamma(f^\sharp(a)) \tag{2.2}$$

In Fig. 2.9 we have a graphical representation of soundness. In particular, Fig. 2.9 (a) refers to the condition $\alpha \circ f(x) \leq_A f^\sharp \circ \alpha(x)$, which compares the computational process in the abstract domain, while Fig. 2.9 (b) refers to the condition $f \circ \gamma(x) \leq_C \gamma \circ f^\sharp(x)$, which compares the results of the computations on the concrete domain. Given a concrete operation $f : C \to C$, we can order the correct approximations of f with respect to (C, α, γ, A): let $f_1^\sharp$ and $f_2^\sharp$ be two correct approximations of f in A, then $f_1^\sharp$ is a better approximation of $f_2^\sharp$ if $f_1^\sharp \sqsubseteq f_2^\sharp$. Hence, if $f_1^\sharp$ is better than $f_2^\sharp$, it means that, given the same input, the output of $f_1^\sharp$ is more precise than the one of $f_2^\sharp$. It is well known that, given a concrete function $f : C \to C$ and a Galois connection (C, α, γ, A), there exists a *best correct approximation* of f on A, usually denoted as f^A. In fact, it is possible to show that $\alpha \circ f \circ \gamma : A \to A$ is a correct approximation of f on A, and that for every correct approximation $f^\sharp$ of f we have that: $\forall x \in A : \alpha(f(\gamma(x))) \leq_A f^\sharp(x)$, i.e., $\alpha \circ f \circ \gamma \sqsubseteq f^\sharp$. Observe that the definition of best correct approximation only depends upon the structure of the underlying abstract domain, namely the best correct approximation of any concrete function is uniquely determined by the Galois connection (C, α, γ, A). For example, consider the concrete square operation on $\wp(\mathbb{Z})$ introduced earlier, and the abstract operation $sq^\sharp$ which is an approximation of the square function on the abstract domain *Sign* defined following the rule of signs. It is clear that this provides a sound approximation of the square function, that is $\forall x, y \in Sign : sq(\gamma(x)) \subseteq sq^\sharp(\gamma(x))$.

Completeness

When the concrete and abstract processes of calculus preserve the same precision, i.e., when soundness is satisfied with equality, we say that the abstract

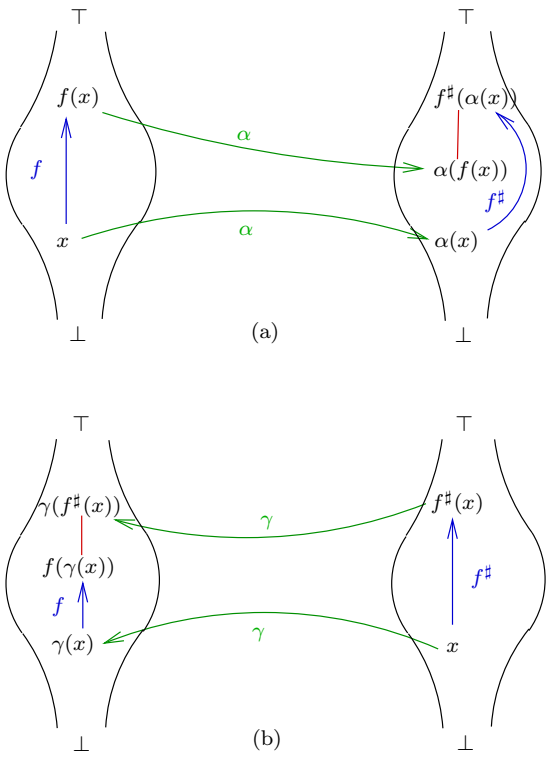


Fig. 2.9. Soundness

function is a *complete* approximation of the concrete one. The equivalent soundness conditions (2.1) and (2.2) introduced above can be strengthened to two different (i.e., incomparable) notions of *completeness*.

Definition 2.25. Given a Galois connection (C, α, γ, A) and a concrete function $f : C \to C$ and an abstract function $f^\sharp : A \to A$ then:

- if $\alpha \circ f = f^\sharp \circ \alpha$ the abstract function $f^\sharp$ is *backward-complete* for f;
- if $f \circ \gamma = \gamma \circ f^\sharp$ the abstract function $f^\sharp$ is *forward-complete* for f.

Both backward ($\mathcal{B}$) and forward ($\mathcal{F}$) completeness encode an ideal situation where no loss of precision arises in abstract computations: $\mathcal{B}$-completeness considers abstractions on the output of operations while $\mathcal{F}$-completeness considers abstractions on the input to operations. For example, $sq^\sharp$ is $\mathcal{B}$-complete for sq on *Sign* while it is not $\mathcal{F}$-complete because $sq(\gamma(0+)) = \{x^2 \in \mathbb{Z} \mid x > 0\} \subsetneq \{x \in \mathbb{Z} \mid x > 0\} = \gamma(sq^\sharp(0+))$. Also, observe that $plus^\sharp$ is neither backward nor forward complete for *plus* on *Sign*. Moreover, observe that the abstract domain *Sign* is not $\mathcal{B}$-complete for addition, in fact $\alpha(\{3,5\} + \{-2,0\}) = \alpha(\{1,2,3,5\}) = 0+$ while $\alpha(\{3,5\}) \oplus \alpha(\{-2,0\}) = 0+ \oplus 0- = \mathbb{Z}$. In Fig. 2.10 (a) we provide a

graphical representation of $\mathcal{B}$-completeness, while Fig. 2.10 (b) represents the $\mathcal{F}$-completeness case. The two notions of completeness can be expressed in terms of closure operators, in particular:

- $\rho \in uco(\wp(C))$ is $\mathcal{B}$-complete for f if $\rho \circ f = \rho \circ f \circ \rho$;
- $\rho \in uco(\wp(C))$ is $\mathcal{F}$-complete for f if $f \circ \rho = \rho \circ f \circ \rho$.

Clearly, when ρ is both $\mathcal{B}$ and $\mathcal{F}$ complete for f, then ρ is a morphism $f \circ \rho = \rho \circ f$.

While any abstract domain A induces the so-called canonical best correct approximation, not all abstract domains induce a $\mathcal{B}$ ($\mathcal{F}$)-complete abstraction. However, if there exists a complete function for f on the abstract domain $\alpha(C)$, then $\alpha \circ f \circ \gamma$ is also complete and viceversa [43]. This means that it is possible to define a complete function for f on $\alpha(C)$ if and only if $\alpha \circ f \circ \gamma$ is complete [61].

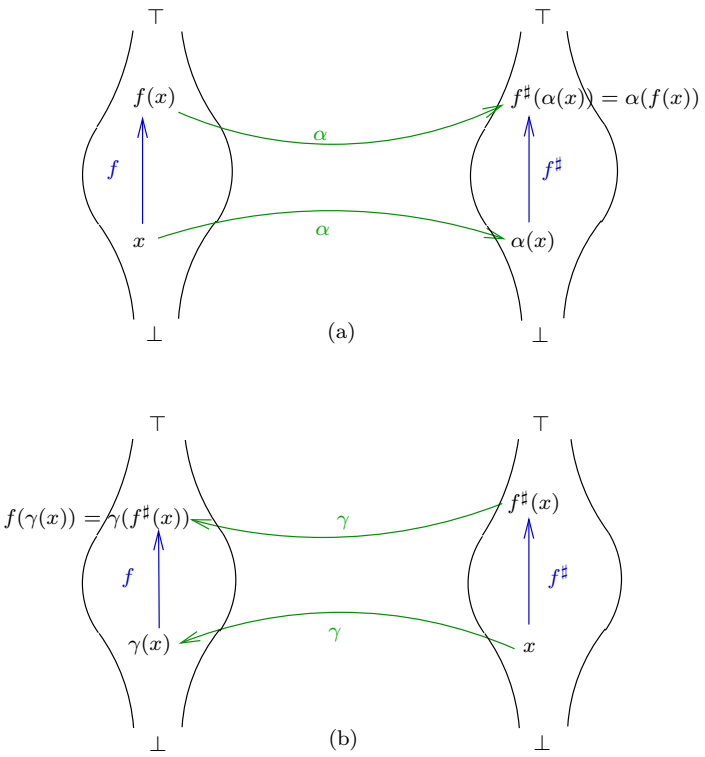


Fig. 2.10. Completeness

Completeness Refinements

It turns out that both $\mathcal{B}$ and $\mathcal{F}$-completeness are abstract domain properties, namely they only depend on the structure of the underlying abstract domain, in the sense that the abstract domain A determines whether it is possible to define a backward or forward complete operation $f^\sharp$ on A [60, 61]. Let us introduce a family of domain transformers that make an abstract domain complete. These transformations, defined in terms of a function f on the concrete domain C, transform an abstract domain A, namely a closure operator, in order to make it complete as regards function f adding the smallest possible amount of information. Thus, these transformers are obtained by finding the most abstract domain that contains A and that is complete for f, generally called *complete shell* of A. Observe that completeness can be obtained also by erasing form A the minimal amount of information in order to make it complete (*complete core* of A). In this thesis we only consider complete shells. The following result gives the basis for the definition of a systematic method for minimally refining a domain in order to make it complete for a given function.

Theorem 2.26. [60, 61] Let $f : C \to C$ be continuous and $\rho \in uco(C)$. Then:

- ρ is $\mathcal{B}$-complete for f iff $\bigcup_{y \in \rho(C)} max(f^{-1}(\downarrow y)) \subseteq \rho(C)$;
- ρ is $\mathcal{F}$-complete for f iff $\forall x \in \rho(C).f(x) \in \rho(C)$.

This means that $\mathcal{B}$-complete domains are closed under maximal inverse image of the function f, while $\mathcal{F}$-complete domains are closed under direct image of f. Let us consider domain transformations that allow to minimally transform any abstract domain A, not complete for f, in order to get completeness.

Definition 2.27. [60] Let C be a complete lattice and $f : C \to C$ be a continuous function. We define $R_f^{\mathcal{B}}, R_f^{\mathcal{F}} : uco(C) \to uco(C)$ such that:

- $R_f^{\mathcal{B}} \stackrel{\text{def}}{=} \lambda X \in uco(C).\mathcal{M}(\bigcup_{y \in X} max(f^{-1}(y)))$;
- $R_f^{\mathcal{F}} \stackrel{\text{def}}{=} \lambda X \in uco(C).\mathcal{M}(f(X))$.

It is clear that $R_f^{\mathcal{B}}$ is monotone on $uco(C)$, because f is monotone on the complete lattice $\langle \wp(C), \subseteq \rangle$. Moreover, by definition, $R_f^{\mathcal{B}}(X) \sqsubseteq X$. The definition of $R_f^{\mathcal{B}}$ follows the idea that the inverse image of f contains all the elements that make a domain backward complete for f. On the other side, also $R_f^{\mathcal{F}}$ is monotone and $R_f^{\mathcal{F}}(X) \sqsubseteq X$. Analogously, the definition of $R_f^{\mathcal{F}}$ follows the idea that the image of f contains all the elements that make a domain forward complete. Observe that an abstract domain A is $\mathcal{B}$-complete for f if and only if $A \sqsubseteq R_f^{\mathcal{B}}(A)$, and analogously A is $\mathcal{F}$-complete for f if and only if $A \sqsubseteq R_f^{\mathcal{F}}(A)$. These observations allow us to build the $\mathcal{B}(\mathcal{F})$-complete domain as a fixpoint. In particular we have the following result.

Theorem 2.28. [60, 61] Consider a closure $\rho \in uco(C)$ and assume that it is not backward complete neither forward complete with regard to the concrete function $f : C \to C$.

– The backward complete shell of ρ is given by:

$$\mathcal{R}_f^{\mathcal{B}}(\rho) = gfp^{\sqsubseteq}\lambda\varphi.\rho \sqcap R_f^{\mathcal{B}}(\varphi)$$

– The forward complete shell of ρ is given by:

$$\mathcal{R}_f^{\mathcal{F}}(\rho) = gfp^{\sqsubseteq}\lambda\varphi.\rho \sqcap R_f^{\mathcal{F}}(\varphi)$$

Therefore, given a continuous function $f : C \to C$ and an abstract domain $A \in uco(C)$, the more abstract domain which includes A and is $\mathcal{B}(\mathcal{F})$-complete for f is respectively $\mathcal{R}_f^{\mathcal{B}}(A)$ and $\mathcal{R}_f^{\mathcal{F}}(A)$.

For example, it turns out that the backward complete shell of the abstract domain *Sign* with respect to addition is given by the abstract domain of *Interval* [61], namley $\mathcal{R}_+^{\mathcal{B}}(Sign) = Interval$. In fact, as observed above $\alpha_{Sign}(\{3,5\} + \{-2,0\}) \neq \alpha_{Sign}(\{3,5\}) \oplus \alpha_{Sign}(\{-2,0\})$, whereas for intervals $\alpha_I(\{3,5\} + \{-2,0\}) = \alpha_I(\{1,2,3,5\}) = [1,5]$ and $\alpha_I(\{3,5\}) \oplus \alpha_I(\{-2,0\}) = [3,5] \oplus [-2,0] = [1,5]$.

2.2.3 Abstract Semantics

As observed earlier, one interest of abstract interpretation theory is the systematic design of approximate semantics of programs. Consider a Galois connection (C, α, γ, A) and the concrete semantics $\mathbf{S}$ of programs $\mathbb{P}$ computed on the concrete domain C. As usual, the semantics obtained replacing C with one of its abstractions A, and each function F defined on C with a corresponding correct approximation $F^\sharp$ on A, is called the abstract semantics. The abstract semantics $\mathbf{S}^\sharp$, as well as abstract functions, has to be correct with respect to the concrete semantics $\mathbf{S}$, that is for every program $P \in \mathbb{P}$, $\alpha(\mathbf{S}[\![P]\!])$ has to be an approximation of $\mathbf{S}^\sharp[\![P]\!]$. Let us consider the concrete semantics $\mathbf{S}[\![P]\!]$ of program P given, as usual, in fixpoint form $\mathbf{S}[\![P]\!] = lfp^{\leq_C} F[\![P]\!]$, where the semantic transformer $F : C \to C$ is monotonic and defined on the concrete domain of objects C. The abstract semantics $\mathbf{S}^\sharp[\![P]\!]$ can be computed as $lfp^{\leq_A} F^\sharp$, where $F^\sharp = \alpha \circ F \circ \gamma$ is given by the best correct approximation of F in A. In this case soundness is guaranteed, namely $\alpha(lfp^{\leq_C}(F)) \leq_A lfp^{\leq_A} F^\sharp$, i.e., $\alpha(\mathbf{S}[\![P]\!]) \leq_A \mathbf{S}^\sharp[\![P]\!]$. Thus, a correct approximation of the concrete semantics $\mathbf{S}$ can be systematically derived by computing the least fixpoint of the best correct approximation of F on the abstract domain A. As usual, completeness of the abstract semantics is not always guaranteed. The following well known result (see e.g. [5, 43]) states that if the abstract domain A is $\mathcal{B}$-complete for the monotone function $F : C \to C$, then the abstract semantics is complete as well.

Theorem 2.29. [FIXPOINT TRANSFER] *Given a Galois connection* (C, α, γ, A)*, and a concrete monotone function* $F : C \to C$*, if* $\alpha \circ F = F^\sharp \circ \alpha$ *(resp.* $\alpha \circ F \leq_A F^\sharp \circ \alpha$*) then* $\alpha(lfp^{\leq_C} F) = lfp^{\leq_A} F^\sharp$ *(resp.* $\alpha(lfp^{\leq_C} F) \leq_A lfp^{\leq_A} F^\sharp$*).*

This means that if the abstract domain is $\mathcal{B}$-complete for the semantic transfer F, then the abstract semantics coincides with the abstraction of the concrete semantics, i.e., $\mathbf{S}^\sharp[\![P]\!] = \alpha(\mathbf{S}[\![P]\!])$. Thus, when the abstract domain is $\mathcal{B}$-complete for F the least fixpoint of the best correct approximation of F on A provides a precise, i.e., complete, approximation of the concrete semantics.

2.3 Syntactic and Semantic Program Transformations

A program transformation is a meaning preserving mapping defined on programming languages [125]. Program transformations aim at improving reliability, productivity, maintenance, security, and analysis of software without sacrificing performances. Commonly used program transformations include constant propagation [82], partial evaluation [36, 79], slicing [152], reverse engineering [154], compilation [127], code obfuscation [35] and software watermarking [32]. Investigating the effects of program transformations on program semantics, i.e., studying the corresponding semantic transformations, is a necessary step in order to prove meaning preservation of the syntactic transformations. In this section we recall the recent result of Cousot and Cousot [44], where the authors formally define the relation between syntactic and semantic program transformations in terms of abstract interpretation. In particular the authors provide a language-independent methodology for systematically deriving syntactic program transformations as approximations of the semantic ones (for which is easier to prove meaning preservation).

In the following, syntactic arguments are between double square brackets $[\![...]\!]$ while semantic/mathematical arguments are between round brackets (...). Given the set $\mathbb{P}$ of all possible programs, let $\mathbf{S}[\![P]\!] \in \mathcal{D}$ denote the semantics of program $P \in \mathbb{P}$. The semantic domain $\mathcal{D}$ is a poset $\langle \mathcal{D}, \sqsubseteq \rangle$, where the partial order $\sqsubseteq$ denotes relative precision, i.e., $\mathcal{Q} \sqsubseteq \mathcal{S}$ means that semantics $\mathcal{S}$ contains less information than semantics $\mathcal{Q}$. The semantic ordering $\sqsubseteq$ induces an order $\preccurlyeq$ on the domain $\mathbb{P}$ of programs, where $P \preccurlyeq Q \stackrel{\text{def}}{=} (\mathbf{S}[\![P]\!] \sqsubseteq \mathbf{S}[\![Q]\!])$. Thus, $\langle \mathbb{P}/_{\eqsim}, \preccurlyeq \rangle$ is a poset, and $\mathbb{P}/_{\eqsim}$ denotes the classes of syntactically equivalent programs, where $P \eqsim Q \stackrel{\text{def}}{=} (\mathbf{S}[\![P]\!] = \mathbf{S}[\![Q]\!])$.

According to Cousot and Cousot [44], given a program $P \in \mathbb{P}$, a *syntactic program transformation* $\mathbb{t}$ returns the transformed program $\mathbb{t}[\![P]\!] \in \mathbb{P}$. The effects of $\mathbb{t}$ on program semantics define the corresponding *semantic transformation* t that takes the semantics $\mathbf{S}[\![P]\!]$ of program P, and returns the semantics $\mathbf{S}[\![\mathbb{t}[\![P]\!]]\!]$ of the transformed program. A program transformation $\mathbb{t}$ is *correct* if it is meaning preserving with respect to some *observational abstraction* $\alpha_{\mathcal{O}}$, namely

if $\forall P \in \mathbb{P} : \alpha_{\mathcal{O}}(\mathbf{S}[\![P]\!]) = \alpha_{\mathcal{O}}(\mathbf{S}[\![\mathbb{t}[\![P]\!]]\!])$. Considering programs as abstractions of their semantics leads to the following Galois insertion:

$$\langle \mathcal{D}, \sqsubseteq \rangle \underset{\mathbb{p}}{\overset{\mathbf{S}}{\leftrightarrows}} \langle \mathbb{P}/_{\eqsim}, \trianglelefteq \rangle \tag{2.3}$$

where $\mathbb{p}[\mathcal{S}]$ is the simplest program whose semantics upper approximates $\mathcal{S} \in \mathcal{D}$. Observe that (2.3) is a Galois insertion thanks to the fact that programs are considered up to syntactic equivalence. In fact, given a program $P \in \mathbb{P}$, $\mathbb{p}(\mathbf{S}[\![P]\!]) \eqsim P$ but potentially $\mathbb{p}(\mathbf{S}[\![P]\!])$ may be different from P because of dead code elimination. Thus $\mathbb{p}(\mathbf{S}[\![P]\!])$ and P are syntactically equivalent since they differ only for (potential) dead code that is not present in the semantics.

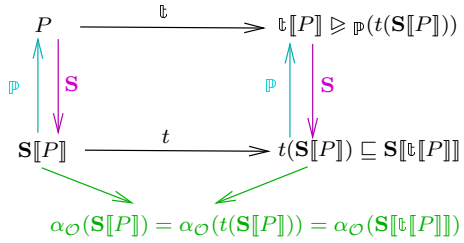


Fig. 2.11. Syntactic-Semantic Program Transformations

The scheme in Fig. 2.11 shows that each semantic transformation induces a syntactic transformation and viceversa:

$$t(\mathbf{S}[\![P]\!]) \stackrel{\text{def}}{=} \mathbf{S}[\![\mathbb{t}[\![\mathbb{p}(\mathbf{S}[\![P]\!])]\!]]\!] \qquad \mathbb{t}[\![P]\!] \stackrel{\text{def}}{=} \mathbb{p}(t(\mathbf{S}[\![P]\!]))$$

In particular, the above equation on the right expresses the fact that a syntactic transformation can be seen as an abstraction of the corresponding semantic transformation. In the following we show how this formalization provides a systematic methodology for designing syntactic transformations from semantic ones. Observe that, when the semantic transformation t relies on undecidable results, any effective algorithm $\mathbb{t}$ is an approximation of the ideal transformation $\mathbb{p} \circ t \circ \mathbf{S}$. This means that, in general, $\mathbb{p}(t(\mathbf{S}[\![P]\!])) \trianglelefteq \mathbb{t}[\![P]\!]$. Considering the Galois insertion (2.3) this constraint corresponds to the correctness condition $t(\mathbf{S}[\![P]\!]) \sqsubseteq \mathbf{S}[\![\mathbb{t}[\![P]\!]]\!]$.

According to Cousot and Cousot [44], in general, program transformation corresponds to a loss of information on program semantics, this approximation is formalized by the following Galois connection:

$$\langle \mathcal{D}, \sqsubseteq \rangle \underset{t}{\overset{\gamma_t}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq \rangle \tag{2.4}$$

Composing Galois connections (2.3) and (2.4) we obtain the Galois connection:

$$\langle \mathbb{P}/_{=}, \preccurlyeq \rangle \underset{\mathbb{t}}{\overset{\gamma_{\mathbb{t}}}{\leftrightarrows}} \langle \mathbb{P}/_{=}, \preccurlyeq \rangle$$

Let us elucidate the steps that lead to the systematic design of $\mathbb{t} \stackrel{\text{def}}{=} \mathbb{p} \circ t \circ \mathbf{S}$ from the semantic transformation t:

Step 1 $\mathbb{p}(t(\mathbf{S}[\![P]\!])) = \mathbb{p}(t(\mathit{lfp}F[\![P]\!]))$, considering as usual program semantics expressed in least fix point form as $\mathbf{S}[\![P]\!] = \mathit{lfp}F[\![P]\!]$;

Step 2 $\mathbb{p}(t(\mathit{lfp}F[\![P]\!])) = \mathbb{p}(\mathit{lfp}\hat{F}[\![P]\!])$, where $\hat{F} \stackrel{\text{def}}{=} t \circ F \circ \gamma_t$ follows from the fixpoint upper approximation theorem considering the abstraction t of (2.4), i.e., $t(\mathit{lfp}F[\![P]\!]) = \mathit{lfp}(t \circ F \circ \gamma_t)[\![P]\!]$ (resp. $\sqsubseteq$ for approximations);

Step 3 $\mathbb{p}(\mathit{lfp}\hat{F}[\![P]\!]) = \mathit{lfp}\mathbb{F}[\![P]\!]$, where $\mathbb{F} \stackrel{\text{def}}{=} \mathbb{p} \circ \hat{F} \circ \mathbf{S}$ follows from the fixpoint upper approximation theorem considering the abstraction $\mathbb{p}$ of (2.3), i.e., $\mathbb{p}(\mathit{lfp}\hat{F}[\![P]\!]) = \mathit{lfp}(\mathbb{p} \circ F \circ \mathbf{S})[\![P]\!]$(resp. $\sqsubseteq$ for approximations);

Step 4 $\mathbb{t}[\![P]\!] \stackrel{\text{def}}{=} \mathit{lfp}\mathbb{F}[\![P]\!]$ (resp. $\preccurlyeq$ for approximations).

Given the fixpoint formalization $\mathit{lfp}\mathbb{F}[\![P]\!]$ of the syntactic transformation, it is possible to design an iterative algorithm on posets satisfying ACC.

Algorithmic Transformations

Let us say that a semantic transformation $t : \mathcal{D} \to \mathcal{D}$ is *algorithmic*, denoted $t \in \mathcal{A}$, if it is induced by a syntactic transformation $\mathbb{t}$, i.e., $t = \mathbf{S} \circ \mathbb{t} \circ \mathbb{p}$, namely if there exists an algorithm whose effects on program semantics are exactly the ones of transformation t.

Definition 2.30. A semantic transformation $t : \mathcal{D} \to \mathcal{D}$ is *algorithmic* if there exists an algorithm $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ such that: $t = \mathbf{S} \circ \mathbb{t} \circ \mathbb{p}$.

It is interesting to observe that the abstract domain $\mathbb{P}$ is $\mathcal{F}$-complete for every concrete (semantic) transformation $t \in \mathcal{A}$. This means that for every algorithmic function t it holds that $t \circ \mathbf{S} = \mathbf{S} \circ \mathbb{t}$.

Lemma 2.31. Considering the Galois insertion $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\mathbb{p}}{\overset{\mathbf{S}}{\leftrightarrows}} \langle \mathbb{P}/_{=}, \preccurlyeq \rangle$ we have that the abstract domain $\mathbb{P}$ is $\mathcal{F}$-complete for every $t \in \mathcal{A}$.

PROOF: Given $t \in \mathcal{A}$, we have to show that $\mathbf{S} \circ \mathbb{p} \circ t \circ \mathbf{S} \circ \mathbb{p} = t \circ \mathbf{S} \circ \mathbb{p}$. Let $\mathcal{X} \in \mathcal{D}$:

$$\begin{array}{lll} \mathbf{S}[\![\mathbb{p}(t(\mathbf{S}[\![\mathbb{p}(\mathcal{X})]\!]))]\!] & = \mathbf{S}[\![\mathbb{p}(\mathbf{S}[\![\mathbb{t}[\![\mathbb{p}(\mathbf{S}[\![\mathbb{p}(\mathcal{X})]\!])]\!]]\!])]\!] & [t = \mathbf{S} \circ \mathbb{t} \circ \mathbb{p},\ t \text{ is algorithmic}] \\ & = \mathbf{S}[\![\mathbb{t}[\![\mathbb{p}(\mathbf{S}[\![\mathbb{p}(\mathcal{X})]\!])]\!]]\!] & [\mathbb{p} \circ \mathbf{S} = id] \\ & = t(\mathbf{S}[\![\mathbb{p}(\mathcal{X})]\!]) & [\mathbf{S} \circ \mathbb{t} \circ \mathbb{p} = t] \end{array}$$

□

In particular, observe that $\mathcal{F}$-completeness means that $t \circ \mathbf{S} = \mathbf{S} \circ \mathbb{t}$, namely that there is no loss of precision between the semantic and syntactic transformation when we compare them on the concrete domain $\mathcal{D}$ of program semantics. This also implies that $\mathbb{t} = \mathbb{p} \circ t \circ S$. Thus, when considering algorithmic semantic transformations, the schema in Fig. 2.11 commutes. In this work we focus on code obfuscation, and we consider semantic obfuscators to be algorithmic transformations, since code obfuscation is, in general, an automatic program transformation. Thus, there exists an algorithm that transforms programs according to the semantic obfuscating transformation. As we will see in Chapter 5 the above methodology provides a systematic way for deriving a possible algorithm.

Programming Language

In the following we introduce the simple imperative language considered in [44], which syntax is reported in Table 2.1.

Syntactic Categories:		**Syntax:**	
$n \in \mathbb{Z}$	(integers)	$E ::=$	$n \mid X \mid E_1 - E_2$
$X \in \mathbb{X}$	(variable names)		
$L \in \mathbb{L}$	(labels)		
$E \in \mathbb{E}$	(integer expressions)		
$B \in \mathbb{B}$	(Boolean expressions)	$B ::=$	`true` $\mid$ `false` $\mid$ $E_1 < E_2 \mid \neg B_1 \mid B_1 \vee B_2$
$A \in \mathbb{A}$	(actions)	$A ::=$	$X := E \mid X :=? \mid B$
$C \in \mathbb{C}$	(commands)	$C ::=$	$L : A \to L'$
$P \in \mathbb{P}$	(programs)	$\mathbb{P} ::=$	$\wp(\mathbb{C})$

Table 2.1. Syntax of the programming language

Given a set S, we use $S_\perp$ to denote the set $S \cup \{\perp\}$, where $\perp$ denotes an undefined value[1]. Let $\mathfrak{D}$ be the semantic domain of variables values. A command at label L has the form $L : A \to L'$, where A is an action and L' the label of the command to be executed next. The `stop` command is $L : \texttt{stop} \eqsim L : \texttt{skip} \to \perp$, and a `skip` command is $L : \texttt{skip} \to L' \eqsim L : \texttt{true} \to L'$. Let $var[\![A]\!]$ denote the set of variables occurring in action A:

$$lab[\![L : A \to L']\!] \stackrel{\text{def}}{=} L \qquad lab[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{C \in P} lab[\![C]\!]$$
$$var[\![L : A \to L']\!] \stackrel{\text{def}}{=} var[\![A]\!] \qquad var[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{C \in P} var[\![C]\!]$$
$$suc[\![L : A \to L']\!] \stackrel{\text{def}}{=} L' \qquad act[\![L : A \to L']\!] \stackrel{\text{def}}{=} A$$

The above basic functions are useful in defining the semantics of the considered programming language, which is described in Table 2.2.

[1] We abuse notation and use $\perp$ to denote undefined values of different types, since the type of an undefined value is usually clear from the context.

Value Domains	
$\mathcal{B}_\perp = \{true, false, \perp\}$	(truth values)
$n \in \mathbb{Z}$	(integers)
$\mathfrak{D}_\perp$	(variable values)
$\rho \in \mathfrak{E} = \mathbb{X} \to \mathfrak{D}_\perp$	(environments)
$\Sigma = \mathbb{C} \times \mathfrak{E}$	(program states)
Arithmetic Expressions	$\mathbf{E} : \mathbb{E} \times \mathfrak{E} \to \mathfrak{D}_\perp$
$\mathbf{E}[\![\mathtt{n}]\!]\rho$	$= n$
$\mathbf{E}[\![X]\!]\rho$	$= \rho(X)$
$\mathbf{E}[\![E_1 - E_2]\!]\rho$	$= \mathbf{E}[\![E_1]\!]\rho - \mathbf{E}[\![E_2]\!]\rho$
Boolean Expressions	$\mathbf{B} : \mathbb{B} \times \mathfrak{E} \to \mathcal{B}_\perp$
$\mathbf{B}[\![\mathtt{true}]\!]\rho$	$= true$
$\mathbf{B}[\![\mathtt{false}]\!]\rho$	$= false$
$\mathbf{B}[\![E_1 \ < \ E_2]\!]\rho$	$= \mathbf{E}[\![E_1]\!]\rho \ < \ \mathbf{E}[\![E_2]\!]\rho$
$\mathbf{B}[\![\neg B]\!]\rho$	$= \neg\mathbf{B}[\![B]\!]\rho$
$\mathbf{B}[\![B_1 \vee B_2]\!]\rho$	$= \mathbf{B}[\![B_1]\!]\rho \vee \mathbf{B}[\![B_2]\!]\rho$
Program Actions	$\mathbf{A} : \mathbb{A} \times \mathfrak{E} \to \wp(\mathfrak{E})$
$\mathbf{A}[\![\mathtt{true}]\!]\rho$	$= \{\rho\}$
$\mathbf{A}[\![X := E]\!]\rho$	$= \{\rho[X := \mathbf{E}[\![E]\!]]\}$
$\mathbf{A}[\![X :=?]\!]\rho$	$= \{\ \rho' \mid \exists z \in \mathbb{Z} : \rho' = \rho[X := z]\ \}$
$\mathbf{A}[\![B]\!]\rho$	$= \{\ \rho' \mid \mathbf{B}[\![B]\!]\rho' = \mathtt{true} \wedge \rho' = \rho\ \}$

Table 2.2. Semantics of the programming language

An *environment* $\rho \in \mathfrak{E}$ is a map from variables in $dom(\rho) \subseteq \mathbb{X}$ to values in $\mathfrak{D}_\perp$, therefore $\rho(X)$ represents the value of variable X. Given $V \subseteq \mathbb{X}$ let $\rho|_V$ denote the restriction of environment ρ to the domain $dom(\rho) \cap V$, while $\rho \setminus V$ denotes the restriction of environment ρ to domain $dom(\rho) \setminus V$. Let $\rho[X := n]$ be the environment ρ where value n is assigned to variable X. Let $\mathfrak{E}[\![P]\!]$ denote the set of environments of program P, namely of those environments whose domain is given by the set of program variables, i.e., $var[\![P]\!]$. A *program state* is a pair $\langle \rho, C \rangle$, where C is the next command that has to be executed in environment ρ. Let $\Sigma \stackrel{\text{def}}{=} \mathfrak{E} \times \mathbb{C}$ denote the set of all possible states, in particular $\Sigma[\![P]\!] \stackrel{\text{def}}{=} \mathfrak{E}[\![P]\!] \times \mathbb{C}$ denotes the set of states of program P. The *transition relation* $\mathbf{C} : \Sigma \to \wp(\Sigma)$ between states specifies, as usual, the set of states that are reachable from a given state:

$$\mathbf{C}(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \{\ \langle \rho', C' \rangle \mid \rho' \in \mathbf{A}[\![act(C)]\!]\rho,\ suc[\![C]\!] = lab[\![C']\!]\ \}$$

A state σ is a final/blocking state when $\mathbf{C}(\sigma) = \varnothing$, let $\mathfrak{T}[\![P]\!]$ denote the set of final/blocking states of program P, in particular $\mathfrak{T}[\![P]\!] = \{\ \langle \rho, C \rangle \mid suc[\![C]\!] \in \mathcal{L}[\![P]\!]\ \}$ where $\mathcal{L}[\![P]\!] \subseteq lab[\![P]\!]$. The transition relation can be specified with respect to a program P, $\mathbf{C}[\![P]\!] : \Sigma[\![P]\!] \to \wp(\Sigma[\![P]\!])$:

$$\mathbf{C}[\![P]\!](\langle \rho, C \rangle) \stackrel{\text{def}}{=} \{\ \langle \rho', C' \rangle \in \mathbf{C}(\langle \rho, C \rangle) \mid \rho, \rho' \in \mathfrak{E}[\![P]\!] \wedge C' \in P\ \}$$

Recall that a *finite maximal execution trace* $\sigma \in \mathbf{S}^n[\![P]\!]$ of program P is a finite sequence $\sigma_0...\sigma_{n-1} \in \Sigma^+$ of states of length n, i.e., $|\sigma| = n$, such that each state σ_i with $i \in [1, n-1]$ is a possible successor of the previous state σ_{i-1}, i.e., $\sigma_i \in \mathbf{C}(\sigma_{i-1})$, and the last state σ_{i-1} is a blocking state. The *maximal finite trace semantics* $\mathbf{S}^+[\![P]\!]$ of program P is given by the union of all finite maximal traces of length $n > 0$, namely $\mathbf{S}^+[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{n>0} \mathbf{S}^n[\![P]\!]$. Observe that $\mathbf{S}^+[\![P]\!]$ can be expressed as the least fixpoint of the monotone function $F^+[\![P]\!] : \wp(\Sigma^+[\![P]\!]) \to \wp(\Sigma^+[\![P]\!])$ defined as follows:

$$F^+[\![P]\!](\mathcal{X}) \stackrel{\text{def}}{=} \mathfrak{T}[\![P]\!] \cup \left\{ \ \sigma_i\sigma_j\sigma \ \middle| \ \sigma_j \in \mathbf{C}[\![P]\!](\sigma_i),\ \sigma_j\sigma \in \mathcal{X} \ \right\}$$

An *infinite execution trace* $\sigma \in \mathbf{S}^\omega[\![P]\!]$ of a program P is an infinite sequence $\sigma_0...\sigma_i... \in \Sigma^\omega$ of length $|\sigma| = \omega$, such that each state σ_{i+1} is a successor of the previous state, i.e., $\sigma_{i+1} \in \mathbf{C}(\sigma_i)$. $\mathbf{S}^\omega[\![P]\!]$ can be computed as the $gfp^{\subseteq} F^\omega[\![P]\!]$, where function $F^\omega[\![P]\!] : \wp(\Sigma^\omega[\![P]\!]) \to \wp(\Sigma^\omega[\![P]\!])$ is defined as:

$$F^\omega[\![P]\!](\mathcal{X}) \stackrel{\text{def}}{=} \left\{ \ \sigma_i\sigma_j\sigma \ \middle| \ \sigma_j \in \mathbf{C}[\![P]\!](\sigma_i),\ \sigma_j\sigma \in \mathcal{X} \ \right\}$$

As usual, the *maximal trace semantics* $\mathbf{S}^\infty[\![P]\!] \in \wp(\Sigma^\infty)$ of program P is given by the union of its finite and infinite traces, namely $\mathbf{S}^\infty[\![P]\!] \stackrel{\text{def}}{=} \mathbf{S}^+[\![P]\!] \cup \mathbf{S}^\omega[\![P]\!]$.

3

Code Obfuscation

In this chapter we introduce the notion of code obfuscation together with the main applications of this technique. In particular, Section 3.1 presents code obfuscation as a promising defense technique against attacks to the intellectual property of software. We provide an overview of the existing technical approaches to software protection, highlighting the advantages of code obfuscation with respect to the other proposed techniques. Next, we introduce the notions of potency, resilience, cost and stealth as parameters for measuring the quality of an obfuscating transformation, followed by an overview of obfuscating techniques, classified according to the taxonomy proposed by Collberg et al. [34]. Then, we report some of the most significant theoretical results on code obfuscation, and we observe how some of these results discourage code obfuscation while others prove its potential. If on the one hand code obfuscation is a promising defense technique, on the other hand it is often used by malware writers, i.e., hackers, to foil malware detectors. Thus, researchers are working on the design of powerful obfuscating transformations and powerful deobfuscation techniques in order to improve both software protection and malware detection. Section 3.2 focuses on the detection of obfuscated malware. We describe the different typologies of malicious programs, classified according to their malicious goal and infection routine. Next, we provide an overview of the techniques used to detect malicious behaviours, with particular attention to signature-based detection algorithms, and we describe how code obfuscation may help malware writers in avoiding detection. Then, we report some of the major theoretical limitations of malware detection. To conclude, we present some more sophisticated techniques for malware detection, such as the ones based on formal methods (e.g., model checking, program slicing and data mining).

3.1 Software Protection

Software protection against malicious host attacks is a key concern in computer industry. Software piracy, malicious reverse engineering and software tampering are the major types of attacks that Bob can use to gain an economic edge over Alice [33]. Assume that Bob has legally purchased an application from Alice. Once Bob has physical access to the application, he can make illegal copies of it and then sell them to ingenuous clients. This attack is known as *software piracy* and refers to the illegal reproduction and distribution of proprietary programs. By decompiling and *(malicious) reverse engineering* Alice's application Bob can extract proprietary algorithms and data structures and incorporate them into his own application. In this way Bob does not recover the entire application, which clearly violates the law [133], but he can still significantly reduce cost and time needed to develop his own software. Assume that Alice's application provides a service for which the client has to pay a certain amount of electronic money. In this case Bob can try to *tamper* with the application in order, for example, to change the amount of money he has to pay or the money destination.

There are legal measures and technical approaches to protect software against these attacks. Legal measures include copyright, patent and license. Copyright laws protect the form in which an idea is expressed but not the idea itself. Thus, software copyright protects a program but not the algorithms and methods within the program. While software copyright protects the code against literal copying, software patent defends also the underlying ideas and the features of the software. Another possibility for the producer to defend his software is to stipulate a contract, called software license, with the client. A software license is typically a complex document that establishes the usage rights that are granted to the client as well as the client limitations. For example, a software license might define a limit on the maximal number of concurrent users of the software, or it might bind the usage of the software to a specific individual. The producer can revoke the license every time that the client violates the contract.

Obtaining patent protection for software is usually expensive and it may be hard for Alice to enforce the law against a larger and more powerful competitor. Moreover, in general, legal protection in one country cannot be extended to other nations. In fact, the Berne Convention (1886) establishes the national treatment of copyright of other countries. This means that a nation, for example France, has to treat each work copyrighted in a different country, for example Italy, as if it was protected by the local copyright law, the french one in the considered example. Thus, a more attractive alternative for Alice is to use technical methods to protect her software. Some early attempts to technical software protection are described in [21, 64, 138].

Software watermarking is a defense technique used to prevent software piracy (e.g., [32, 51, 115]). The idea is for Alice to discourage illegal copying

by embedding a *signature*, i.e., a copyright notice, into her software. When an illegal copy is made, Alice can prove her ownership by extracting her signature from the code. The signature has to be hidden inside the code in such a way that it is difficult for Bob to detect and then remove it. In order to identify the copyright violator, as well as the illegal copies, Alice could insert a different signature, usually called *fingerprint*, in each copy of the application she distributes. In this way the particular signature that Alice extracts from an illegal copy indicates also the guilty client (Bob).

Alice can protect her application against malicious tampering by using *tamper-proofing code*, namely code that is able to detect if Bob has tampered with some sensitive information of the application (e.g., if Bob has changed the amount of electronic money he has to pay to get the service from Alice), and in this case makes the program fail or sends an alert message to Alice [7, 8, 18, 19].

Since any attack to the intellectual property of software starts with a reverse engineering phase, the first defense consists in blocking (or at least delaying) this process. Existing forms of technical software protection to prevent malicious reverse engineering include:

- *Hardware Device*: A typical hardware-based method for software protection is the *dongle*. A dongle is a small hardware device that plugs into the serial or USB port on a computer to ensure that only authorized users can copy or use specific software applications. When a software protected by a dongle runs it checks the dongle for authentication as it is loaded, if the dongle is not present the software refuses to run (or runs in a restricted mode). Dongles are generally used to protect expensive applications, while their employment in the mainstream software market usually meets resistance from users. Moreover, dongles do not provide a complete solution to the malicious host problem. In fact, there are flaws in the existing hardware devices that a malicious user can exploit in order to bypass protection [65]. For example, a malicious user could exploit the weaknesses in the communication protocol between the dongle and the protected software in order to gain complete access to the application even when the dongle is not present.
- *Server-side execution*: Alice sells her *services* rather than her application. The user connects to Alice's site and runs the program remotely paying a small amount of electronic money every time. In this way, even if Bob purchases the services from Alice, he never has physical access to the application and he cannot reverse engineer it. The obvious disadvantage of this technique is performance degradation due to network communication, limited bandwidth and latency, and to the load on the server when many clients try to access it during a short period of time. When only some parts of the application are regarded as proprietary by Alice it is not necessary to protect the entire application. Thus, the application can be broken into a private part, which

executes remotely, and a public part, which runs locally on the user's site. *Partial server side execution* may limit performance degradation.

- *Encryption*: Alice gives to Bob an *encrypted* version of her application. Unless decryption takes place in hardware, it will be possible for Bob to interpret and decrypt compiled code. Hence, this technique works only if the decryption/execution process takes place in hardware. Hardware decryption systems have been described in [72]. The idea is to have a co-processor (cryptochip) that decrypts instructions before execution. In this way the decrypted code is never accessible to Bob, and the degree of security depends on the scheme used to encrypt the code. In general, different platforms need distinct circuits to interface with the cryptochip. Therefore this approach is unsuitable when the application has to run on many different platforms.
- *Obfuscation*: Alice *obfuscates* the program before distributing it. Code obfuscation consists in syntactically transforming a program in such a way that the obfuscated program becomes more difficult to understand, i.e., to reverse engineer, while maintaining its functional behaviour. Thus, the idea of code obfuscation is to make a program so difficult to understand that reverse engineering it becomes uneconomical in terms of resources and time. However, code obfuscation cannot fully protect an application against a malicious reverse engineering attack. In fact, given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application. Software watermarking techniques usually perform some sort of code obfuscation in order to protect the inserted signature form Bob. Thus, code obfuscation is often used to enforce software watermarking.

Defenses based on hardware devices and encryption have the drawback of requiring special hardware, while server side execution suffers from network overhead. Thus, code obfuscation seems to be more appropriate when dealing with mobile programs. This is one of the reason way, in recent years, code obfuscation has attracted researchers interest in preventing malicious reverse engineering, leading to the design of different obfuscating transformations (e.g., [29, 31, 33, 35, 100, 123, 148]).

The reverse engineering process intends to recover the original source code from the machine code. It typically begins with a disassembly phase, which translates machine code to assembly code, then followed by a number of decompilation steps, that try to recover source (or high level) code from assembly code (see Fig 3.1). Thus, in order to complicate reverse engineering, we can either confuse the disassembly or the decompilation phase. Decompilation mainly consists of performing a static analysis of the assembly code, including data-flow, control-flow and type analysis. Therefore, a program transformation that obstructs such static analyses acts as an obfuscating technique. Most of the existing obfuscat-

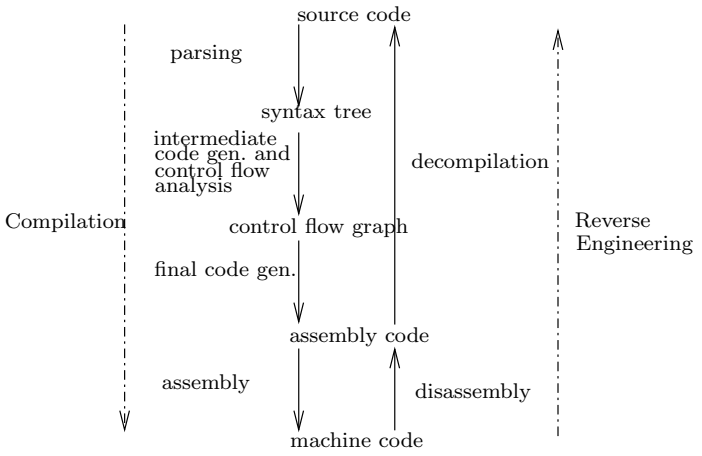


Fig. 3.1. Compilation and Reverse Engineering [100]

ing transformations focus on the decompilation phase (e.g. [31, 34, 35, 123, 148]), while less attention has been paid to obstruct the disassembly process. However, recently, some work has been done in the direction of obfuscating executable code in order to thwart well-known static disassembly techniques, such as linear sweep and recursive traversal [100]. Obstructing correct disassembly can be achieved also by changing repeatedly the program code while it executes [103].

3.1.1 Obfuscating Transformations and their Evaluation

An *obfuscator* is a program that transforms programs in such a way that the transformed (obfuscated) code is functionally equivalent to the original one but more difficult to understand. This means that the *observable behaviour*, i.e., the behaviour as experienced by the user, of the two programs must be identical. In the following we recall the general definition of obfuscating transformations introduced by Collberg et al. [31, 34, 35].

Definition Let $\mathbb{t} : \mathbb{P} \rightarrow \mathbb{P}$ be a program transformation from a source program P into a target program P'. $\mathbb{t} : \mathbb{P} \rightarrow \mathbb{P}$ is an *obfuscating transformation*, if:

- the transformation $\mathbb{t}$ is *potent* and
- P and P' have the same *observable behaviour*, i.e., if P fails to terminate or it terminates with an error condition then P' may or may not terminate, otherwise P' must terminate and produce the same output as P.

A program transformation is potent if the transformed (obfuscated) program is more complex to understand than the original one. It is clear that the above

definition of code obfuscation relies on the notion of *potency* of a transformation, and therefore on a fixed metric for measuring program complexity, which is a quite hard problem [69, 109]. In the literature there are a lot of different metrics for program complexity, that can be used according to the current need. For example, the complexity of a program can be measured by: the length of the program (the number of instructions and arguments) [69], the nesting level (the number of nested conditions) [70], the data flow (the number of references to local variables) [124], or the data structure complexity (the complexity of the data structures declared in the program) [77]. Given a metric for program complexity it is possible to measure the potency of a transformation, namely how much more difficult is the transformed program to understand than the original one. It is clear that, in order to design a good obfuscator, the potency of the transformation has to be maximized.

While the potency of an obfuscating transformation measures how much obscurity has been added to a program, the *resilience* of a transformation measures how difficult it is to break for an automatic deobfuscator. Resilience takes into account both the amount of time required to construct a deobfuscator and the execution time and space actually required by the deobfuscator. Some highly resilient obfuscating transformations are one-way transformations, in the sense that they can never be undone. This because one-way transformations usually remove information (e.g., formatting removal, scramble variable names) from the program. In general, other obfuscations have different degrees of resilience, depending on how difficult it is to identify and remove the useless information that has been added by the obfuscation. A good obfuscator tries to maximize its resilience.

Another important factor to take into account when designing an obfuscating transformation is the execution time/space penalty added to program execution by the obfuscation. The *cost* of an obfuscating transformation measures the computational overhead added to the obfuscated program with respect to the original one. Some trivial obfuscations (e.g., scrambling identifiers) incur no runtime cost, while most of the commonly used obfuscating transformations cause a varying amount of overhead. It is clear that, in practice, there would be a threshold identifying the limit between the acceptable/unacceptable amount of penalty caused by obfuscating transformations. In fact, there is often a tradeoff between the level of obscurity that can be added to a program and the transformation cost.

Another useful measure is the *stealth* of a transformation. An obfuscating transformation is stealthy if it does not "stand out" from the rest of the program, namely if the obfuscated code resembles the original code as much as possible. It is clear that stealth is a context-sensitive notion, meaning that what is stealthy in one program may not be stealthy in another one. If the obfuscating transformation introduces code widely different from the original code, it is easy

for a reverse engineer to detect and remove the obfuscation. For this reason, a good obfuscator has to insert stealthy code.

Obfuscating transformations are usually evaluated and compared with respect to their potency, resilience, cost and stealth. The problem with these quality metrics is that they are difficult to measure precisely. For example, potency, resilience and stealth of an obfuscating transformation often present some kind of statistical properties and their measure clearly depends on the personal skills of the programmer that is trying to break the transformation.

3.1.2 A Taxonomy of Obfuscating Transformations

Obfuscating transformations can be classified according to the kind of information they target [34]. In the following we briefly present the main classes of this taxonomy together with some examples.

Layout obfuscators Layout obfuscating transformations act on code information that is unnecessary to its execution (used by the Java obfuscator Crema [146]). These obfuscations are typically trivial and reduce the amount of information available to a human reader. Layout transformations include the removal of comments and the change of identifiers. For example, by replacing identifiers of methods and variables with meaningless identifiers, any information on the functionality of a method or on the role of a variable is removed. Scrambling identifier names is a one-way transformation that adds no penalty during execution.

Data obfuscators Data obfuscators operate on program data structures and they can be further classified according to the kind of operation they perform on data. *Storage and encoding* transformations affect how data is stored in memory and the methods used to interpret stored data [31]. An example of encoding transformation consists in replacing an integer variable i by $i' = 8 \times i + 3$ and then modifying the instructions involving variable i in order to preserve program functionality (e.g., int $i = 1$; while $(i < 1000)$... becomes int $i = 11$; while $(i < 8003)$...). In this case there is a trade-off between resilience and potency on one hand, and cost on the other hand. For example, the encoding proposed above, $i' = 8 \times i + 3$ adds little extra execution time but it can be deobfuscated using common compiler analysis techniques. *Aggregation* obfuscations alter how data are grouped together, making it more difficult for a reverse engineer to restore the program's data structure. These transformations can split, fold or merge arrays in order to complicate the access to arrays, for example by transforming a two-dimensional array in a one-dimensional array and viceversa. These transformations have an high potency since they introduce structures where there was originally none, or they remove structures from the

original program [34,157]. *Ordering* transformations change how data is ordered. For example, they can reorder arrays using a function $f(i)$ to determine the position of the i-th element of the array, while the i-th element is usually stored in the i-th position of the array. These transformations have low potency while their resilience is one-way [157].

Control code obfuscators Control obfuscations attempt to confuse the program control flow. These transformations can affect either the aggregation, the ordering or the computations of the program control flow. *Aggregation* transformations change the way in which program statements are grouped together, by splitting and merging fragments of code. For example, it is possible to *inline* procedures, that is, replacing a procedure call with the statements of the called procedures itself. A very useful companion transformation to inlining is *outlining*, which aggregates code that does not belong together, for example turning a sequence of statements into a procedure. Another class of control aggregation obfuscations are loop transformations, such as *loop unrolling* which replicates the body of the loop one or more times. These transformations have a low resilience when applied in isolation, while their resilience grows significantly when these transformations are combined together. A program is easier to understand if logically related items are also physically close in the source text. Following this observation, *ordering* transformations attempt to randomize, when possible, the placement of any item in the source text (e.g., reordering of independent statements). For example, in certain cases, it is possible to reorder loops by running them backwards (loop reversal). These transformations have usually low potency but their resilience is high. *Computation* transformations insert new (redundant or dead) code in order to hide the real control flow behind statements that are irrelevant. For example it has been observed that there is a strong correlation between the perceived complexity of a piece of code and the numbers of predicates it contains. Thus, these control transformations often rely on the existence of *opaque predicates*, that is, predicates whose value is known a priori to the obfuscation, but it is difficult for the deobfuscator to deduce. By inserting these opaque predicates, it is possible to break up the original control flow of a program. In this case the resilience (resp. stealthy) of the transformation depends on the resilience (resp. stealthy) of the opaque predicate, namely on how difficult it is to detect the inserted opaque predicate (resp. on how different the inserted opaque predicate is form the rest of the code).

Opaque Predicates For transformations that alter the program control flow, a certain amount of computational overhead would be unavoidable. Opaque predicates are often used to design control code obfuscating transformations that are cheap and resilient to attacks from deobfuscators. Control flow obfuscation by mean of opaque predicates was introduced by Collberg et al. [35]. An opaque

predicate is a predicate whose constant value is known at obfuscation time, but it is hard for a deobfuscator to deduce this value from automated program analysis. Fig. 3.2 illustrates the different types of opaque predicates, where solid lines indicate paths that may sometimes be taken and dashed lines paths that will never be taken. Typically P^T denotes a true opaque predicate, namely a

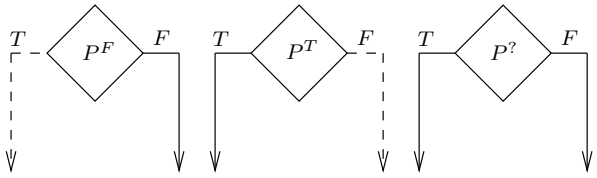


Fig. 3.2. Opaque Predicates

predicate that always evaluates to *true*, P^F a false opaque predicate, that is a predicate that always evaluates to *false*, and $P^?$ an unknown opaque predicate, namely a predicate that sometimes evaluates to *true* and sometimes evaluates to *false*. Consider, for example, the insertion of a branch instruction controlled by a true opaque predicate P^T. In this case the *true* path starts with the next action of the original program, while the *false* path leads to termination or buggy code. This confuses the attacker who is not aware of the always *true* value of the opaque predicate and has to consider both paths. It is clear that this transformation does not affect program functionality since at run time P^T is always evaluated *true* and therefore the *true* path is the only one to be executed. While the insertion of false opaque predicates is analogous to one of true opaque predicates, the case of unknown opaque predicates is slightly different. When a branch instruction is controlled by an unknown opaque predicate $P^?$ both the *false* and *true* path have to be equivalent to the sequence of original program actions. In fact $P^?$ may evaluate either *true* or *false* and in both cases program functionality has to be preserved.

In order to deobfuscate a program an attacker usually employs various static and dynamic analysis techniques. Thus, it seems natural to construct opaque predicates on problems that are hard to handle by such analyses. For example, Collberg et al. [35] show how to construct opaque predicates based on the difficulty of alias analysis [75, 130]. Their idea is to add to the obfuscated program code that constructs a complex dynamic structure and that maintains a set of pointers into this structure. These pointers can be updated but they have to preserve certain invariant (e.g., two pointers never have to refer to the same location). Hence, it is possible to design opaque predicates that need a precise alias analysis of the dynamic structure to be broken. Another possibility is to design opaque predicates based on the difficulty of analyzing parallel

programs with respect to sequential ones. In this case a global data structure is created and occasionally updated by concurrently executing sequences of instructions (threads when dealing with Java) [35]. Once again, it is possible to design opaque constructs based on such dynamic structure.

More recently Palsberg et al. [126] have introduced the notion of *dynamic opaque predicate* as a possible improvement over static opaque predicates presented above. The idea is to define a family of correlated predicates which evaluate to the same value in any single program run, but this value might vary over different program runs. This notion of dynamic opaque predicate has then been extended to *temporary unstable* or *distributed* opaque predicates in a distributed environment [108]. The value of a temporary unstable opaque predicate may change in different program points during the same run of the program. The idea is that the opaque predicate value depends on predetermined embedded message communication patterns between different processes that maintain the opaque predicate. Two are the main advantages of using temporary unstable opaque predicates: re-usability and resilience against static analysis attacks and dynamic monitoring (see [108] for details).

In [15] it has been proposed a general notion of opacity, where a property over program executions is said to be opaque if it is not possible to deduce it for an observer. The authors show how different security notions, including non-interference and anonymity, can be guaranteed by the opacity of certain properties on program executions. Moreover, the authors observe that certify the opacity of a certain property is in general undecidable and they propose a technique for approximating the original notion of opacity in order to make it decidable. Other similar/further works on this general theory for opacity exists (e.g., [16, 90]).

It is interesting to observe that opaque predicates find interesting applications not only in control code obfuscation techniques, but also in data obfuscation techniques [31], software watermarking [116] and tamper proofing [126].

In general, software protection through code obfuscation is obtained by combining many different obfuscating transformations. Which transformations is better to apply to a certain application and the order in which transformations should be applied are two main concerns when constructing an obfuscating tool. These problems have been addressed in [29] where the authors propose a possible solution.

3.1.3 Positive and Negative Theoretical Results

Many researchers recognise that one major drawback of existing code obfuscating techniques is the lack of a rigorous theoretical background allowing one to study and compare different obfuscating transformations. In fact, a formal

definition of obfuscating transformations together with a precise model for the attackers performing the deobfuscation process are necessary in order to provide formal proofs of the effectiveness of different obfuscating techniques with respect to attackers. The relative scarcity of theoretical papers on code obfuscation suggests that this is still an open research area. Thus, it is not surprising if in the existing literature it is possible to find inconsistencies in definitions, models and conclusions. In the following we briefly recall some of the most significant existing theoretical results on code obfuscation.

Wang et al. observe that any intelligent tampering attack requires knowledge of the program semantics, usually obtained by static analysis. Thus, they provide a code obfuscation technique based on control flow flattering and variable aliasing that drastically reduces the precision of static analysis [147, 148]. The basic idea of Wang et al. is to make the analysis of the program control flow dependent on the analysis of the program data flow, and then to use aliasing to complicate data flow analysis. In particular, the proposed obfuscation transforms the original control flow of the program into a flattened one where each basic block can be the successor/predecessor of any other basic block. The actual program control flow is determined dynamically by a dispatcher. At the end of each basic block the dispatcher variable is changed through complicated pointer manipulations, making control flow analysis depend on complex data flow analysis. The authors provide a proof of the resilience of their obfuscation technique, and such proof relies on the difficulty of determining precise indirect branch target addresses of dispatchers in presence of aliased pointers.

However, this approach is restricted to the case of intra-procedural analyses. A software obfuscation technique, related to the one of Wang et al. and based on obstructing inter-procedural analysis and on the difficulty of alias analysis is proposed in [123], together with a theoretical proof of its effectiveness. Another promising theoretical result considers an obfuscation technique based on the insertion of hard combinatorial problems with known solution into the program using semantic preserving transformations. Chow et al. [22] claim that this obfuscating transformation makes the deobfuscation process PSPACE-complete.

Another novel and formal approach to code obfuscation is the one of Drape [53]. Drape observes that it is difficult to provide proofs of the fact that a given obfuscation preserves program behaviour. In this work, the author provides a formal framework for reasoning and proving the preservation of the observational behaviour of some data obfuscation techniques. In particular, Drape proposes to obfuscate abstract data-types and to view obfuscation as a data refinement. The data-type operations used to obfuscate are modeled as functional programs making it more easy to construct the corresponding proofs. The proposed framework has been applied to some data-types as for example lists, sets, trees and matrices [53, 54].

These results suggest the possibility of a significant increase in the difficulty of reverse engineering through code obfuscation.

In contrast, a well known negative theoretical result on code obfuscation is given by Barak et al. [11], who show that code obfuscation is impossible. This result seems to prevent code obfuscation at all. However, this result is stated and proved in the context of a rather specific model of code obfuscation. Barak et al. [11] define an obfuscator as a program transformer $\mathcal{O}$ satisfying the following conditions: (1) $\mathcal{O}(P)$ is functionally equivalent to P, (2) the slowdown of $\mathcal{O}(P)$ with respect to P is polynomial both in time and space, and (3) anything that one can compute from $\mathcal{O}(P)$ can also be computed from the observation of the input-output behavior of P. Hence, this formalizes an "ideal" obfuscator, where the original and obfuscated program have identical behaviours (1,2) and where the obfuscated program is unintelligible to an adversary (3). In practical contexts these constraints can be relaxed. In particular, in [33–35, 123, 148] the authors consider a number of obfuscating transformations that make the obfuscated program significantly slower or larger than the unobfuscated one. These proposals even allow the obfuscated program to have different side-effects than the original one, or not to terminate when the original program terminates with an error condition. The only requirement they make is that the *observable behaviour* — namely the behaviour observed by a generic user — of the two programs should be identical. Besides, many researchers are interested in transformations that raise the difficulty of reverse engineering a program, even if they do not make it impossible as request by point (3) of the Barak's definition. In fact, an obfuscating transformation that requires a very expensive analysis, in terms of resources and time, to be undone, protects the intellectual property of proprietary software by making reverse engineering of the obfuscated programs uneconomical [73]. Moreover, the "ideal" obfuscator of Barak et al. has to be able to protect *every* program. In fact the impossibility of code obfuscation is proved by providing a contrived class of functions that are not obfuscatable. It would be interesting to understand to which portion of programs of practical interest this negative result can be applied.

Relaxing the constraints of Barak's definition, it is reasonable and of practical interest to study the possibility of obfuscating, i.e., making more difficult to understand, significant programs. Moreover, some of the authors of the impossibility result have later achieved some positive results on code obfuscation [102], that, together with the works of Canetti and Wee [17, 151], show, under certain assumptions, how to obfuscate classes of functions of practical interest. On the other hand, another negative theoretical result, related but even stronger than the impossibility result, has been proved in [63]. This result enforces the notion of obfuscation of Barak et al. and it is therefore susceptible to the same limitations.

3.1.4 Code Deobfuscation

In order to evaluate the resilience of obfuscating transformations, we have to consider the techniques generally used by a reverse engineer, i.e., the deobfuscation tools available. Deobfuscation techniques are usually based either on static or on dynamic analysis. While static program analysis is performed without executing the program, dynamic analysis takes place at run time. Common static analysis techniques include detection of dead code and uninitialized variables, program slicing [152], alias analysis [92], partial evaluation [36,79], and data flow analysis [71]. Dynamic analysis is performed by testing the program on sample input data, since it is infeasible to test all possible program control paths due to combinatorial explosion. Static analysis is conservative, meaning that the properties deduced by static deobfuscating techniques are weaker than the ones that may actually be true (i.e., this corresponds to an over-approximation). This guarantees soundness, although the inferred properties may be so weak to be useless. On the other hand, a dynamic analysis precisely observes only a subset of all possible execution paths of a program (i.e., this corresponds to an under-approximation). Recent work on combining static and dynamic program analysis seems to provide a set of heuristics for disclosing some significant obfuscating techniques [144].

There are few preliminary works on deobfuscation and reverse engineering complexity. It has been shown that data disassembly and decompilation is undecidable in the case of binary code [100]. On the other hand, Appel proved that, under specific and restrictive conditions, deobfuscation is an NP-easy problem [4].

As observed earlier code obfuscation cannot fully protect an application against a malicious reverse engineering attack. In fact, given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application. Thus, the power of code obfuscation relies in the possibility of delaying the release of confidential information for a sufficiently long time [73]. Once again, the aim of code obfuscation is to confuse the program in such a way that reverse engineering it becomes uneconomical.

3.2 Malware Detection

A *malware* is a program with a malicious intent that has the potential to harm, without the user informed consent, the machine on which it executes or the network over which it communicates. The growing size and complexity of modern information systems, together with the growing connectivity of computers through the Internet have promoted the widespread propagation of malicious code [110]. The term *payload* refers to the action that a malicious program is designed to perform on the infected machine. Malware are usually classified

according to their propagation method and their payload into the following categories [110].

- *Viruses*: A virus is a self-propagating program that attaches itself to host programs and propagates when an infected program executes. A virus typically consists of an infection procedure, that searches for a new program to infect, and of an injure procedure, that performs the virus payload (usually when a certain condition is satisfied). Some viruses are designed to damage the machines by corrupting programs, deleting files, or reformatting the hard disk. Other viruses, usually called benign viruses, simply replicate themselves. However, also benign viruses compromise the machines, typically by occupying memory space used and needed by legitimate programs.
- *Worms*: A malicious program that uses a network to send copies of itself to other systems is usually called a computer worm. Unlike viruses, worms do not need an host program to carry them around but rather propagate across a network. A typical example of this class of malicious programs are email worms that arrive as email, where the message body or attachment contains the worm code, and spread through email messages. In general, worms do not contain a specific payload but they are only designed to spread. However, the growth in network traffic and other unintended effects are usually causes of major disruption.
- *Trojan horses*: As viruses, Trojan horses hide their malicious intent inside host programs that may look useful, or at least harmless, to an unsuspecting user. Trojan horses can be either corrupted legitimate programs that execute malicious code when they run, or standalone programs that masquerade as something else in order to obtain the user unaware complicity needed to accomplish their goals. In fact, Trojan horses are characterized by their dependency on actions from the victims, who have at least to run the malicious code. In order to tempt the user to install such malicious programs, Trojan horses usually look like something innocuous or desirable (as in the myth).
- *Back-doors*: A back-door is a computer program designed to bypass local security policies in order to allow external entities to have remote control over a machine or a network. Back-doors can either be standalone programs that are able to avoid casual inspection, or corrupted versions of legitimate programs.
- *Spyware*: The term spyware usually refers to malicious programs designed to monitor users' actions in order to collect private information and send them to an external entity over the Internet. Spyware, for example, try to intercept passwords or credit cards numbers. More generally, a spyware is any program that subverts users' operations for the benefit of a third party. Observe that there are many innocuous spyware that observe and collect information for benign purposes, for example for advertisement.

Very harmful attacks can be constructed by combining malicious programs of different classes. Consider for example a worm with a payload that installs a new back-door. Every time the worm replicates and infects new machines it installs a back-door. This provides an easy an fast way to gain remote access to a growing number of hosts (the infected ones). Despite their differences, every malicious program exploits some system or network security vulnerability in order to infect and damage new victims.

3.2.1 Detection Techniques

If, on one hand, the malware detection problem, also known as intrusion detection problem, has attracted researchers attention as an interesting and challenging problem (e.g. [23, 24, 111, 140]), on the other hand malware writers, i.e., hackers, have become more and more clever. As malware detectors improve, being able to identify the latest and more sophisticated malware, the hackers invent new methods for evading detection. This co-evolution has lead to the design of very sophisticated malware and detection algorithms [119]. Intrusion detection is concerned with the identification of activities that have been generated with the intention of compromise data or machines [3]. In particular, malware detectors analyze a program (or data) in order to identify activities that may be indicative of a malicious attack. When this happens the malware detector alerts the administrator who will handle the situation. Let us briefly present two major approaches to malware detection, known as anomaly detection and misuse detection.

Anomaly detection

This approach is also known as *profile-based intrusion detection* or *statistical intrusion detection*. It assumes that malicious code will cause behaviours different from the ones normally observed in a system. In fact, anomaly detection is based on the definition of "normality" and classifies as malicious any activity that deviates from it [111]. It observes the "normal" activities of the user and then creates behaviour profiles that represent the threshold that divides normal from abnormal behaviours. Such profiles can be modeled using statistical-based [87, 96], rule-based [145] and immunology-based methods [58]. It is clear that false negatives, i.e., classification of illicit activity as benign, and false positives, i.e., classification of legitimate activity as malicious, arise due to the imprecision of the definition of normal behaviour. In fact, classifying what is normal is a difficult task and involves technical factors as well as some sort of knowledge from expert users.

Disadvantages of anomaly detection: One of the main drawback of anomaly detection is that abnormal behaviours are not always a sign of malware infection.

This may lead to false alarms, i.e., false positives, that report intrusion even if it has not occurred [2,97]. In fact, systems often exhibit legitimate but previously unseen behaviours, which leads anomaly detection techniques to produce a high degree of false alarms. Another problem is that a clever attacker could induce the anomaly detection system to accept anomalous, i.e., malicious, behaviours as normal ones by corrupting the system during the training phase [56]. Moreover, in general, modeling normal behaviors is a complicate and computationally complex task [2,10].

Advantages of anomaly detection: Anomaly detection has the advantage that no specific knowledge of malicious code is required in order to detect infection. Thus, it may potentially discover attacks that have not been seen before [88]. In fact, any activity that differs from the normal behaviour is considered for further analysis despite what has been previously classified as malicious.

Misuse detection

This detection system is also known as *signature-based detection* or *pattern-based detection*. Misuse detection assumes that attacks can be described through patterns, and every time an occurrence of those patterns is found it is classified as a potential intrusion [9,111,140]. These systems monitor attacks in order to identify signatures that contain information distinctive to a specific attack. In fact, signatures are usually sequences of instructions or events characterizing a known malicious behaviour [89,136]. Sometimes signatures can express the distribution of particular actions in a program, in this case we speak of frequency-based signatures. Thus, a signature is a pattern that captures the essence of an attack and that can be used to identify the attack when it occurs [122]. It is clear that this technique relies on a list of signatures, traditionally known as *signature databases* [114]. Hence, a key point of this approach is the generation of signatures that correctly represent the essence of a malicious behaviour [111]. If the signatures are too specific, misuse detection may not recognise slight variations of an attack, while signatures that are too flexible may lead to a great amount of false alarms.

Disadvantages of misuse detection: The main disadvantage of signature-based detection is the fact that this systems are not able to detect "new" attacks, namely attacks for which a signature has not been produced. Signature database needs to be frequently updated in order to deal with novel kinds of attacks. Generating signatures is a time consuming and error prone task and requires a high level of expertise [10,111], and researchers have lately concentrated on automatic signature generation techniques (e.g. [14,83,99,120,121,153]).

Advantages of misuse detection: The reason for the widespread deployment of signature-based detection systems is their low false positive rate and ease of use. In fact, misuse detection techniques do not consume as much resources as

anomaly detection systems.

The fact that misuse detection and anomaly detection have advantages that complement each other, has lead to the development of detection system that combine the two approaches. These hybrid systems [131, 136] rely on attack patterns for signature-based detection and, at the same time, they implement learning and profile algorithms to identify invalid actions [2, 78].

However, misuse detection and anomaly detection have their own limitations with no clear solutions up to know [111]. Hence, current research on intrusion detection focuses on ad-hoc techniques for different applications. This approach turns out to be impractical due to the advancement of Internet and the consequent growth in the application scenarios for intrusion detection. Thus, we do agree with McHugh, who claims that *"further significant progress for intrusion detection will depend on the development of an underlying theoretical basis for it"* [111]. Recently attempts to develop such theoretical basis can be found in [98].

To conclude we mention another common technique for intrusion detection which is known as *specification-based detection*. These techniques monitor programs execution and claim the presence of a malware (or intrusion) when they detect deviations from programs original behaviours [85, 86]. Thus, they rely on program specifications that describe the intended behaviour of (uninfected) programs. Specification-based detection systems are similar to anomaly detection in that they also detect attacks as deviations from a norm. The main difference being that they are based on manually developed specifications that capture legitimate systems' behaviours, and not on machine learning techniques. One of the main drawback of these techniques is the high cost of the development of detailed specification, for which an high level of expertise is often needed. This technique, as well as anomaly detection, has the potential of detecting previously unseen attacks.

In this thesis we are particularly interested in investigating and improving signature-based detection techniques (see Chapter 6). For this reason, in the following, we describe the major countermeasures that hackers have implemented to avoid signature-based detection.

3.2.2 Metamorphic Malware

In order to deal with advanced detection systems malware writers recur to better hiding techniques. This co-evolution of defense and attacks techniques has lead to the development of polymorphic and metamorphic malware.

Polymorphic malware: Polymorphic malware change their syntactic representation, usually by encrypting the malicious payload and decrypting it during execution. In particular, they use different encryption methods (often randomly generated) to encrypt the constant part of the malicious code every time they infect a new machine [118, 141]. Such malware avoid detection until the means of decryption has been discovered (sometimes inefficiencies in the randomness of the polymorphic engine may provide an easy solution). Another possibility for dealing with polymorphic malware consists in executing the possibly infected program on a virtual computer, where the malware cannot cause damage, and look at the original malware body produced at run time by the decryption routine. In fact, once decrypted, all generated polymorphic malware look alike, and standard signature-based detection schemes can be used.

Metamorphic malware: Metamorphic malware employ a more powerful technique to avoid detection. The idea is that each successive generation of a malware modifies the syntax while leaving the semantics unchanged. As observed in the previous section, code obfuscation is a program transformation that changes the way in which a program is written but not its semantics. Thus, it is not surprising that attackers have resorted to program obfuscation for evading malware detection. Of course, attackers have the choice of creating new malware from scratch, but that does not appear to be a favored tactic [139]. Program obfuscation transforms a program, either manually or automatically, by inserting new code or modifying existing code in order to make understanding and detection harder, at the same time preserving malicious behaviour. It is clear that obfuscating transformations can easily defeat signature-based detection mechanisms. For example, if a signature describes a certain sequence of instructions [140], then those instructions can be reordered or replaced with equivalent instructions [155, 156]. Such obfuscations are especially applicable on CISC architectures, such as the Intel IA-32 [76], where the instruction set is rich and many instructions have overlapping semantics. Moreover, if a signature describes a certain distribution of instructions in the program, insertion of junk code [80, 141, 156] can defeat frequency-based signatures. In order to deal with metamorphic malware, misuse detection should keep an updated database of signatures of all possible malware variations. This is not an easy task, since, in principle, there is an unlimited number of possible mutations.

In the following we consider a fragment of the Chernobyl/CIH virus, designed to infect Windows 95/98/NT executables files [132], together with one of its metamorphic (obfuscated) version. This example is taken from [23]. We report both the binary and the assembly code of the virus, where $(*)$ denotes the instructions added by the transformation.

Original code	
E8 00000000	`call 0h`
5B	`pop ebx`
8D 4B 42	`lead ecx, [ebx + 42h]`
51	`push ecx`
50	`push eax`
50	`push eax`
0F01 4C 24 FE	`sidt [esp - 02h]`
5B	`pop ebx`
83 C3 1C	`add ebx, 1Ch`
FA	`cli`
8B 2B	`mov ebp, [ebx]`

Obfuscated code		
E8 00000000	call 0h	
5B	`pop ebx`	
8D 4B 42	`lead ecx, [ebx + 42h]`	
90	`nop`	(∗)
51	`push ecx`	
50	`push eax`	
50	`push eax`	
90	`nop`	(∗)
0F01 4C 24 FE	`sidt [esp - 02h]`	
5B	`pop ebx`	
83 C3 1C	`add ebx, 1Ch`	
90	`nop`	(∗)
FA	`cli`	
8B 2B	`mov ebp, [ebx]`	

Table 3.1. Original and obfuscated code from Chernobyl/CIH

The following table reports the two different signatures needed by misuse detection schemes to deal with the different versions of the virus.

Signature for the original code
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B

Signature for the obfuscated code
E800 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B

Table 3.2. Signatures

As observed in [150] the metamorphic malware phenomena is not confined to a particular programming language. In fact, in every Turing-complete programming language there is some redundancy, meaning that the mapping from syntax to semantics is many-to-one. This redundancy is at the basis of metamorphic malware, that can evade detection by generating syntactic variants at run time.

There is strong evidence that commercial malware detectors are susceptible to common obfuscation techniques used by malware writers. For example, it has been proved that malware detectors cannot handle obfuscated versions of worms [24], and there are a numerous obfuscating techniques designed to avoid detection (e.g., [55, 81, 129, 140]). Thus, *an important requirement of a robust malware detection technique is to handle obfuscating transformations.*

3.2.3 Theoretical Limitations

An introduction to theoretical computer virology can be found in the early work of Cohen [26]. In particular, Cohen proposes a formal definition of computer virus based on the Turing's model of computation, and proves that precise virus detection is undecidable [27], namely that there is no algorithm that can reliably detect all viruses. Cohen shows also that the detection of evolutions of viruses from normal viruses is undecidable [28], namely that metamorphic malware detection is undecidable. These results have been obtained following a reasoning similar to the one used to prove the undecidability of the Halting Problem [143]. A related undecidability result is the one of Chess and White [20], who prove the existence of a virus type that cannot be detected. Adleman [1] is another researcher who has applied formal computability theory to viruses and viruses detection, showing that the problem is quite intractable.

Despite these results, proving that, in general, viruses detection is impossible, it is possible to develop ad-hoc detection schemes that work for specific viruses (malware).

The (simpler) problem of detecting a mutation of a known finite-length virus has been recently considered. It turns out that the problem of reliably identifying a bounded-length mutating virus is NP-complete [137].

With the advent of polymorphic and metamorphic malware, the malware detection community has begun to face these theoretical limits and to develop detection systems based on formal methods of program analysis.

3.2.4 Formal Methods Approaches

In this section we give a brief presentation of some of the existing approaches to malware detection based on formal methods. We do agree with Lakhoita and Singh, who state that *"formal methods for analysing programs for compilers and verifiers when applied to anti-virus technologies are likely to produce good results for the current generation of malicious code"* [91].

Program Semantics: Christodorescu and Jha [25] observe that the main deficiency of misuse detection is its purely syntactic nature, that ignores the meaning of instructions, namely their semantics. Following this observation, they propose an approach to malware detection that considers the malware semantics, namely the malware behaviour, rather than its syntax. Malicious behaviour is described through a template, namely a generalization of the malicious code that expresses the malicious intent while eliminating implementation details. The idea is that a template does not distinguish between irrelevant variants of the same malware obtained through obfuscation processes. For example, a template uses symbolic variable/constants to handle variable and register renaming, and it is related to

the malware control flow graph in order to deal with code reordering. Then, they propose an algorithm that verifies if a program presents the template behaviour, using some unification process between program variables/constants and malware symbolic variables/constants. This detection approach is able to handle a limited set of obfuscations commonly used by malware writers.

Static Analysis: Bergeron et al. propose a malware detection scheme based on the detection of suspicious system call sequences [12]. In particular, they consider a reduction (subgraph) of the program control flow graph, which contains only the nodes representing certain system calls. Next they check if such subgraph presents known malicious sequences of system calls.

Christodorescu and Jha [23] describe a malware detection system based on language containment and unification. The malicious code and the possibly infected program are modeled as automatons (using unresolved symbols and placeholders for registers to deal with some sorts of obfuscations). In this setting, a program presents a malicious behaviour if the intersection between the language of the malware automaton and the one of the program automaton is not empty.

Model Checking: Singh and Lakhotia [135] specify malicious behaviours through a formula in linear temporal logic (LTL), and then use the model checker SPIN to check if this property is satisfied by the control flow graph of a suspicious program.

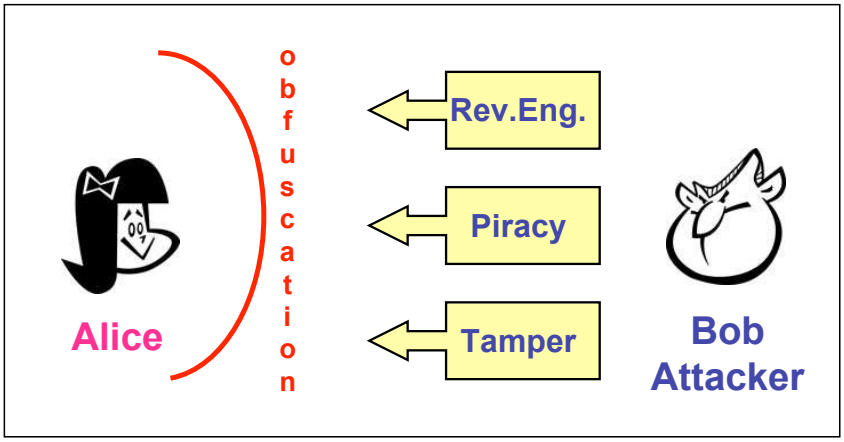
Kinder et al. [84] introduce a new temporal logic CTPL (Computation Tree Predicate Logic), which is an extension of the branching time temporal logic CTL, that takes into account register renaming, allowing a succinct and natural presentation of malicious code patterns. They develop a model checking algorithm for CTPL that, checking if a program satisfies a malware property expressed by a CTPL formula, verifies if the program is infected by the considered malicious behaviour.

Model checking techniques have recently been used also in worm quarantine applications [13]. Worm quarantine techniques seek to dynamically isolate the infected population from the population of uninfected systems, in order to fight malware infection.

Program Slicing: Lo et al. [101] develop a programmable static analysis tool, called MCF (Malicious Code Filter), that uses program slicing and flow analysis to detect malicious code. Their approach relies on tell-tale signs, namely on program properties that characterize the maliciousness of a program. MCF slices the program with respect to these tell-tale signs in order to get a smaller program segment that might perform malicious actions. These segments are further analyzed in order to determine the existence of a malicious behaviour.

Data Mining: Data mining techniques try to discover new knowledge in large data collections. In particular, data mining identifies hidden patterns and trends that a human would not be able to discover efficiently on large databases, employing, for example, machine learning and statistical analysis methods. Lee et al. [93–95] have studied ways to apply data mining techniques to intrusion detection. The basic idea is to use data mining techniques to identify patterns of relevant system features, describing program and user behaviour, in order to recognise both anomalies and known malicious behaviours.

Code Obfuscation as Semantic Transformation



The Malicious Host Perspective

Following a standard definition, an obfuscator is a potent program transformation that preserves the observational behaviour of programs, where potent means that the obfuscated program is more difficult to understand, i.e., more complex, than the original one [35]. If on one side obfuscating transformations aim at confusing some information, on the other side they must preserve program behaviour (i.e., program semantics) to some extent. Even if obfuscating techniques are semantic preserving transformations, the lack of a complete formal setting where these transformations can be studied prevents any possibility of comparing them with respect to their ability to obfuscate properties of program behaviour (i.e., semantic properties). One of the main problem here is to fix a metrics for program complexity. We have seen that, usually, syntactic (textual) measures are considered, such as code length, nesting levels, fan-in-out

complexity, branching, etc. [34]. Semantics-based measures are instead less common, even thought they may provide a deeper insight into the true potency of code obfuscation. In order to understand program complexity from a semantic point of view, we need a formal model for attackers, i.e., for code deobfuscation techniques. Reverse engineering usually begins with a static program analysis of the program. Recently, it has been shown that efficient deobfuscation techniques can be obtained by combining static and dynamic analysis [144]. It is well known that static analysis can be completely and fully specified as an abstract interpretation, i.e., as an approximation, of concrete program semantics [41], while dynamic analysis can be seen as a possibly non decidable approximation of the concrete semantics. Thus, when dealing with static and dynamic attackers syntactic measures of program complexity can be misleading. More significant measures have to be derived from semantics and this, as far as we know, is an open problem.

In this chapter we face this problem by providing a theoretical framework, based on program semantics and abstract interpretation, where formalizing, studying and relating existing code obfuscation transformations with respect to their potency and resilience to attacks. As noticed above, code obfuscation aims at obstructing static or dynamic analysis which can both be expressed as approximations of concrete program semantics. In this sense, code obfuscation can be seen as a way to prevent that some information about program behaviour is disclosed by an abstract interpretation of its semantics. This observation naturally leads us to consider abstract interpretations of concrete program semantics as a formal model for attackers, and obfuscations as semantic transformations. In Section 2.3 we have presented the recent result of Cousot and Cousot, who formalize the relation between syntactic and semantic transformations in the abstract interpretation field, where programs are seen as abstractions of their semantics [44]. This result allows us to relate, in the abstract interpretation framework, each syntactic transformation (code obfuscation) to its semantic counterpart and vice versa. In this setting, the lattice of abstract interpretations provides the right framework to compare attackers by comparing abstractions. This leads us to introduce a semantics-based definition of potency of obfuscating transformations.

Requiring, as usual, obfuscating transformations to preserve input-output (denotational) semantics of programs, seems to be an unreasonable restriction: Semantics at different level of abstractions can be related by abstract interpretation in a hierarchy of semantics [40]. Thus, in general, a program transformation $\mathbb{t}$ which preserves a given semantics in the hierarchy, acts as an obfuscator with respect to the properties that are not preserved by $\mathbb{t}$. The idea is that a program transformation $\mathbb{t}$ is potent if there exists a semantic property, i.e., a semantic abstraction, that is not preserved by $\mathbb{t}$. In this setting, every program transformation can be characterized in terms of the most concrete property it preserves

on the concrete semantics. This mapping of code transformations to the lattice of abstract interpretations allows us to measure and compare the potency of different obfuscating transformations. The idea is that, the more abstract is the most concrete property preserved by a transformation the more potent the transformation is, namely the bigger is the amount of obscurity added by the transformation. In order to characterize the obfuscating behaviour of each program transformation $\mathbb{t}$, we provide a systematic methodology for deriving the most concrete property that is preserved by a given $\mathbb{t}$. This leads to a semantics-based definition of code obfuscation, introduced in Section 4.2, where a program transformation is a $\mathcal{Q}$-obfuscator if: (1) $\mathcal{Q}$ is the most concrete property preserved by the transformation, and (2) there is a not empty set of properties that are not preserved, i.e., obfuscated, by the transformation characterized in terms of $\mathcal{Q}$. We show that this definition of obfuscation is a generalization of the standard notion of code obfuscation (see Theorem 4.5). This is witnessed by the fact that, in principle, following our definition, *any program transformation may potentially act as a code obfuscation.* As an example of our claim, in Section 4.4, we study the obfuscating behaviour of the well known transformation performing constant propagation. The results presented in this chapter has been published in [48].

4.1 Standard Definition of Code Obfuscation

As observed earlier, code obfuscation is a potent program transformation that preserves the observational behaviour of programs. More formally, code obfuscation has been defined as follows.

Definition 4.1. [31, 34, 35] A program transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is an *obfuscator* if:

1. transformation $\mathbb{t}$ is potent and
2. P and $\mathbb{t}[\![P]\!]$ have the same observational behavior, i.e., if P fails to terminate or it terminates with an error condition then $\mathbb{t}[\![P]\!]$ may or may not terminate; otherwise $\mathbb{t}[\![P]\!]$ must terminate and produce the same output as P.

Point 2 of the above definition requires the original and obfuscated program to behave equivalently in case of termination, namely on the maximal finite traces, while no constraints are specified for infinite traces, i.e., in the case of non termination or error. This means that in order to classify a program transformation $\mathbb{t}$ as an obfuscation, we have to analyze the behaviour of the corresponding semantic transformation $t = \mathbf{S} \circ \mathbb{t} \circ \mathbb{p}$ only on finite traces terminating with a final/blocking state. Thus, we can focus only on finite traces, considering the domain Σ^+ of maximal finite trace semantics instead of the more concrete domain

Σ^∞. In the following, a semantic transformation t is called a (semantic) obfuscator in the sense of Definition 4.1, if t is induced by a syntactic transformation $\mathbb{t}$ that is an obfuscator according to the above definition. This is because, as observed earlier, semantic obfuscations, being the semantic counterpart of code obfuscation, are algorithmic transformations. Recall that the maximal finite trace semantics, also known as *angelic* semantics, computed on Σ^+ can be formalized as an abstraction of the maximal trace semantics computed on Σ^∞ [40]. In particular the angelic semantics is obtained by approximating sets of possibly finite or infinite traces with the set of finite traces only, i.e., $\alpha^+ : \wp(\Sigma^\infty) \to \wp(\Sigma^+)$ is defined as $\alpha^+(\mathcal{X}) \stackrel{\text{def}}{=} \mathcal{X} \cap \Sigma^+ = \mathcal{X}^+$, while $\gamma^+ : \wp(\Sigma^+) \to \wp(\Sigma^\infty)$ is given by $\gamma^+(\mathcal{Y}) \stackrel{\text{def}}{=} \mathcal{Y} \cup \Sigma^\omega$. Thus, we have the adjunction shown in Fig. 4.1.

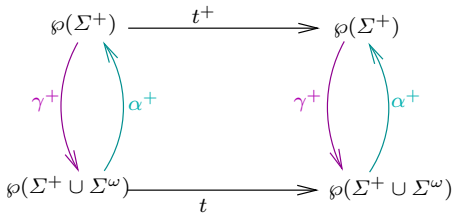


Fig. 4.1. $t^+ = \alpha^+ \circ t \circ \gamma^+$

The following result shows that the preservation of the observational behaviour (point 2 of Definition 4.1) can be equivalently verified on $t : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ or on its best correct approximation $t^+ = \alpha^+ \circ t \circ \gamma^+ : \wp(\Sigma^+) \to \wp(\Sigma^+)$.

Proposition 4.2. The semantic transformation $t : \wp(\Sigma^\infty) \to \wp(\Sigma^\infty)$ preserves the observational behaviour if and only if $t^+ : \wp(\Sigma^+) \to \wp(\Sigma^+)$ does, where $t^+ \stackrel{\text{def}}{=} \alpha^+ \circ t \circ \gamma^+$.

PROOF: Observe that $[t(\mathbf{S}^\infty[\![P]\!]) \cap \Sigma^+ = t^+(\mathbf{S}^+[\![P]\!])]$ since t^+ behaves as t on finite traces:

$$
\begin{aligned}
& t \text{ preserves the observational behaviour} \\
\Leftrightarrow\ & \forall \sigma \in \mathbf{S}^\infty[\![P]\!] \text{: } \sigma \in \Sigma^+,\ \exists \eta \in t(\mathbf{S}^\infty[\![P]\!]) \text{: } \eta \in \Sigma^+,\ \sigma_0 = \eta_0, \sigma_f = \eta_f \\
\Leftrightarrow\ & \forall \sigma \in \mathbf{S}^\infty[\![P]\!] \cap \Sigma^+,\ \exists \eta \in t(\mathbf{S}^\infty[\![P]\!]) \cap \Sigma^+ \text{: } \sigma_0 = \eta_0, \sigma_f = \eta_f \\
\Leftrightarrow\ & \forall \sigma \in \mathbf{S}^+[\![P]\!],\ \exists \eta \in t(\mathbf{S}^\infty[\![P]\!]) \cap \Sigma^+ \text{: } \sigma_0 = \eta_0, \sigma_f = \eta_f \\
& [t(\mathbf{S}^\infty[\![P]\!]) \cap \Sigma^+ = t(\mathbf{S}^+[\![P]\!]) = t^+(\mathbf{S}^+[\![P]\!])] \\
\Leftrightarrow\ & \forall \sigma \in \mathbf{S}^+[\![P]\!],\ \exists \eta \in t^+(\mathbf{S}^+[\![P]\!]) \text{: } \sigma_0 = \eta_0, \sigma_f = \eta_f \\
\Leftrightarrow\ & t^+ \text{ preserves the observational behaviour}
\end{aligned}
$$

□

From now on, t refers to a semantic transformation of sets of finite traces, namely $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$. Recall that in [44] it has been observed that, given a finite trace semantics, i.e., a set of finite traces, it is always possible to derive a program transformation by collecting all the commands along such traces. This is formalized by function $\mathbb{p}^+ : \wp(\Sigma^+) \to \wp(\mathbb{C})$ that maps set of traces into sets of commands according to the following definition:

$$\mathbb{p}^+(\mathcal{X}) \stackrel{\text{def}}{=} \left\{ C \,\middle|\, \exists \sigma \in \mathcal{X} : \exists i \in [0, |\sigma|[: \exists \rho \in \mathfrak{E} : \sigma_i = \langle C, \rho \rangle \right\}$$

From now on we consider the following specification of the Galois connection (2.3) defining the relation between programs and their semantics, where the concretization map is given by the semantic function $\mathbf{S}^+$ and the abstraction map by $\mathbb{p}^+$:

$$\langle \wp(\Sigma^+), \subseteq \rangle \underset{\mathbb{p}^+}{\overset{\mathbf{S}^+}{\leftrightarrows}} \langle \mathbb{P}/_{=}, \subseteq \rangle \tag{4.1}$$

Fig. 4.2 summarizes the observations done so far on code obfuscation and specifies the relation between syntactic and semantic obfuscating transformations, where $t \circ \mathbf{S}^+ = \mathbf{S}^+ \circ \mathbb{t}$.

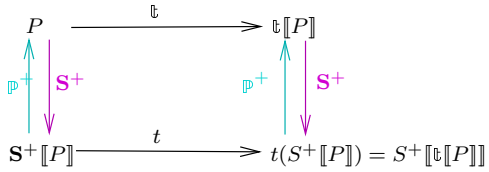


Fig. 4.2. Semantic and syntactic obfuscation

4.2 Semantics-based Definition of Code Obfuscation

As noticed above, one major drawback of existing code obfuscation techniques is the weakness of their theoretical basis, that makes it difficult to formally study and certify their effectiveness. Our idea is to face this problem by providing a theoretical framework, based on program semantics and abstract interpretation, where formalizing, studying and relating code obfuscating transformations with respect to their potency and resilience to attacks.

If on one side obfuscating transformations attempt to mask the program behavior in order to confuse the attacker, on the other side they must preserve the observational behaviour of programs. Preservation of the observational behaviour is guaranteed by the preservation of the denotational semantics *DenSem*,

i.e., by the preservation of the input-output behavior of program executions. Recall that program semantics formalizes program behaviour for every possible input. The set of all program traces, i.e., the maximal trace semantics, expressing the evolution of program states during every possible computation, is a possible formalization of program behaviour, namely a possible program semantics. In the literature there exists many different program semantics. The most common ones include the big-step, termination and non-termination, Plotkin's natural, Smyth's demonic, Hoare's angelic relational and corresponding denotational, Dijkstra's predicate transformer weakest-precondition and weakest-liberal precondition and Hoare's partial and total axiomatic semantics. In [40] Cousot defines a hierarchy of semantics, where the above semantics are all derived by successive abstractions from the maximal trace semantics – also called concrete semantics is the following. In this framework $uco(\wp(\Sigma^\infty))$ represents the lattice of abstract semantics, namely each closure in $uco(\wp(\Sigma^\infty))$ expresses an abstraction of maximal trace semantics. Consider for example the *(natural) denotational semantics DenSem* that abstracts away the history of the computation by observing the input/output relation of finite traces and the input of diverging computations only. It is clear that *DenSem* can be formalized as an abstract interpretation of the maximal trace semantics, in fact $DenSem(X)$ is equal to:

$$\left\{ \sigma \in \Sigma^+ \,\middle|\, \exists \delta \in X^+.\, \sigma_0 = \delta_0 \wedge \sigma_f = \delta_f \right\} \cup \left\{ \sigma \in \Sigma^\omega \,\middle|\, \exists \delta \in X^\omega . \sigma_0 = \delta_0 \right\}$$

where $X^+ \stackrel{\text{def}}{=} X \cap \Sigma^+$ and $X^\omega \stackrel{\text{def}}{=} X \cap \Sigma^\omega$. In this context, considering that only the input/output denotational semantics is preserved as in Definition 4.1 is a restriction on the possible preserved semantics of a program transformation. Our idea is to relax this constraint by providing a definition of code obfuscation which is parametric on the semantic properties to preserve.

We have seen that one of the characterizing features of obfuscating transformation is the fact that they are potent. Thus, in order to give a semantics-based definition of code obfuscation, we need to introduce a definition of program transformation potency based on semantics $\mathbf{S}^+$.

Definition 4.3. A program transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is *potent* if there exists a semantic property $\varphi \in uco(\wp(\Sigma^+))$ and a program $P \in \mathbb{P}$ such that: $\varphi(\mathbf{S}^+[\![P]\!]) \neq \varphi(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!])$.

The idea is that a program transformation $\mathbb{t}$ is potent when there exists a semantic property $\varphi \in uco(\wp(\Sigma^+))$ that is not preserved by $\mathbb{t}$, namely when there exists a property φ obfuscated by $\mathbb{t}$. Given a program transformation $\mathbb{t}$, each semantic property $\varphi \in uco(\wp(\Sigma^+))$ can be classified either as a preserved or as a masked property with respect to $\mathbb{t}$. Thus, in order to distinguish between the properties that are preserved and the ones that are hidden by a transformation $\mathbb{t}$, it is useful to define the most concrete property $\delta_{\mathbb{t}} \in uco(\wp(\Sigma^+))$ preserved

by a given transformation $\mathbb{t}$ on all programs. Let $\{\varphi_i\}_{i\in H}$ be the set of all properties preserved by $\mathbb{t}$, i.e., $\forall i \in H : \forall P \in \mathbb{P} : \varphi_i(\mathbf{S}^+[\![P]\!]) = \varphi_i(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!])$, then $\sqcap_{i\in H}\varphi_i(\mathbf{S}^+[\![P]\!]) = \cap_{i\in H}\varphi_i(\mathbf{S}^+[\![P]\!]) = \cap_{i\in H}\varphi_i(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!]) = \sqcap_{i\in H}\varphi_i(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!])$. Thus, given a program transformation $\mathbb{t}$, there exists an unique most concrete preserved property $\delta_{\mathbb{t}}$. Moreover, property $\delta_{\mathbb{t}}$ can be specified as the least common abstraction between the properties preserved by transformation $\mathbb{t}$ on programs:

$$\delta_{\mathbb{t}} \stackrel{\text{def}}{=} \sqcap \left\{ \varphi \in uco(\wp(\Sigma^+)) \,\middle|\, \forall P \in \mathbb{P} : \varphi(\mathbf{S}^+[\![P]\!]) = \varphi(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!]) \right\}$$

or equivalently:

$$\delta_{\mathbb{t}} \stackrel{\text{def}}{=} \sqcap \left\{ \varphi \in uco(\wp(\Sigma^+)) \,\middle|\, \forall P \in \mathbb{P} : \varphi(\mathbf{S}^+[\![P]\!]) = \varphi(t(\mathbf{S}^+[\![P]\!])) \right\}$$

since we are considering algorithmic transformations where $\mathbf{S}^+ \circ \mathbb{t} = t \circ \mathbf{S}^+$ and therefore φ is preserved by t if and only if it is preserved by $\mathbb{t}$. Given the most concrete property $\delta_{\mathbb{t}}$ preserved by transformation $\mathbb{t}$, we can classify each semantic property $\varphi \in uco(\wp(\Sigma^+))$ either as obfuscated or preserved with respect to $\mathbb{t}$. It is clear that for every $\varphi \in uco(\wp(\Sigma^+))$ such that $\delta_{\mathbb{t}} \sqsubseteq \varphi$, property φ is preserved by transformation $\mathbb{t}$. Moreover, $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi)$ precisely expresses the aspects of property $\varphi \in uco(\wp(\Sigma^+))$ that are obfuscated by transformation $\mathbb{t}$. In fact, the least common abstraction $\delta_{\mathbb{t}} \sqcup \varphi$ represents what the two properties have in common, then by "subtracting" this common part from φ we obtain what transformation $\mathbb{t}$ hides of property φ. Consequently, we say that a property φ is obfuscated by a transformation $\mathbb{t}$ when $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) \neq \top$, namely when something of the property φ has been lost during the transformation $\mathbb{t}$. In fact, if property φ is preserved we have $\delta_{\mathbb{t}} \sqsubseteq \varphi$ and therefore $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) = \top$. Following this observation, we formalize the set of properties that are masked by a program transformation as follows:

$$O_{\delta_{\mathbb{t}}} = \left\{ \varphi \in uco(\wp(\Sigma^+)) \,\middle|\, \varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) \neq \top \right\}$$

The collection $O_{\delta_{\mathbb{t}}}$ precisely identifies the set of properties that are not preserved by transformation $\mathbb{t}$. In fact, $\varphi \ominus (\delta_{\mathbb{t}} \sqcup \varphi) = \top$ if and only if $\varphi = \delta_{\mathbb{t}} \sqcup \varphi$ if and only if $\delta_{\mathbb{t}} \sqsubseteq \varphi$ if and only if φ is preserved by $\mathbb{t}$. Thus, a program transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ can be seen as an obfuscating transformation that preserves all the properties φ such that $\delta_{\mathbb{t}} \sqsubseteq \varphi$ and obfuscates all the properties in $O_{\delta_{\mathbb{t}}}$. Hence, the obfuscating behaviour of a transformation $\mathbb{t}$ can be characterized in terms of the most concrete property it preserves. These observations lead to the following definition of code obfuscation.

Definition 4.4. $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is a δ-*obfuscator* if $\delta = \delta_{\mathbb{t}}$ and $O_\delta \neq \varnothing$.

It is possible to show how our semantics-based definition of code obfuscation provides a generalization of the standard notion of obfuscator by Collberg et al.

reported in Definition 4.1. In fact, given the family $\mathbb{O}$ of program transformations that are classified as obfuscators following Collberg's definition, it turns out that $\mathbb{O}$ corresponds to the set of δ-obfuscators where δ is at least the denotational semantics.

Theorem 4.5. $\mathbb{O} = \left\{ \delta\text{-}obfuscators \,\middle|\, \delta \sqsubseteq DenSem \right\}$.

PROOF: We have to show that $\mathbb{O} = \{\mathbb{t} \mid \delta_{\mathbb{t}} \sqsubseteq DenSem, O_{\delta_{\mathbb{t}}} \neq \varnothing\}$. The condition $O_{\delta_{\mathbb{t}}} \neq \varnothing$ requires transformation $\mathbb{t}$ to be potent, and it is therefore equivalent to point 1 of Definition 4.1. Thus, we have to show that the program transformations that preserve at least the *DenSem* of programs are the ones that preserve the observational behaviour, namely that satisfy point 2 of Collberg's definition.

$$
\begin{aligned}
\mathbb{t} \in \left\{ \mathbb{t} \,\middle|\, \delta_{\mathbb{t}} \sqsubseteq DenSem \right\} &\Leftrightarrow \forall P \in \mathbb{P} : \delta_{\mathbb{t}}(\mathbf{S}^{+}[\![P]\!]) = \delta_{\mathbb{t}}(t(\mathbf{S}^{+}[\![P]\!])) \\
&\Leftrightarrow \forall \sigma \in \mathbf{S}^{+}[\![P]\!],\, \exists \eta \in t(\mathbf{S}^{+}[\![P]\!]) \colon \sigma_0 = \eta_0, \sigma_f = \eta_f \\
&\Leftrightarrow t \text{ preserves the observational behaviour} \\
&\quad [\text{from Prop 4.2}] \\
&\Leftrightarrow \mathbb{t} \text{ preserves the observational behaviour}
\end{aligned}
$$

□

The formalization of the notion of code obfuscation introduced by Definition 4.4 allows us to consider every program transformation as a potential code obfuscator, where the potency of the transformation is expressed in terms of the most precise preserved property. Moreover, it generalizes the standard definition of code obfuscation, where obfuscating transformations are not forced to be *DenSem*-preserving but they can also be more invasive as far as the preserved property maintains enough information with respect to the current need. For example, let us consider an application P that is responsible of keeping updated the total amount *tot* of the bank account of each client, and an application Q that sends a warning to the bad clients every time their total amount *tot* corresponds to a negative value. Assume that we are interested in protecting application P through code obfuscation. It is clear that, in order to ensure the proper execution of application Q, the obfuscated version of application P has to preserve (at least) the sign of variable *tot*. This means that, we can allow obfuscations that loose the observational behaviour of application P but not the sign of variable *tot*. In this setting, a program transformation that replaces the value of variable *tot* with its double $2tot$ is an obfuscation following our definition, while it is not an obfuscation following Collberg's definition.

Moreover, it is clear that our notion of code obfuscation provides a more precise characterization of the obfuscating behaviour of a program transformation $\mathbb{t}$ even when $\mathbb{t}$ satisfies the Collberg's definition. In fact, while the standard notion of obfuscation only distinguish between transformations that preserve *DenSem*

and the ones that do not preserve *DenSem*, our definition of code obfuscation relies on a much finer classification that distinguishes between every possible abstractions of trace semantics.

4.2.1 Constructive characterization of $\delta_{\mathbb{t}}$

As argued above, the most concrete property $\delta_{\mathbb{t}}$, preserved by program transformation $\mathbb{t}$, specifies the obfuscating behaviour of $\mathbb{t}$ by defining the borderline between masked and preserved properties. Thus, in order to view any transformation $\mathbb{t}$ as a potential obfuscation, we need a constructive methodology for deriving the most concrete property preserved by $\mathbb{t}$. We have already observed that obfuscating transformations are algorithmic transformations and therefore $\mathbf{S}^+ \circ \mathbb{t} = t \circ \mathbf{S}^+$. This means that for every property $\varphi \in uco(\wp(\Sigma^+))$, and every program $P \in \mathbb{P}$ we have that $\varphi(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!]) = \varphi(t(\mathbf{S}^+[\![P]\!]))$. This means that a property φ is preserved by program transformation $\mathbb{t}$ if and only if it is preserved by its semantic counterpart $t = \mathbf{S}^+ \circ \mathbb{t} \circ \mathbb{p}$. Thus, when dealing with preserved properties we can equivalently refer to the syntactic or to the semantic transformation. In this section we consider the semantic transformation since we find it more convenient.

Given a program $P \in \mathbb{P}$ and a semantic transformation $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$, we can define a domain transformer $K_{P,t} : uco(\wp(\Sigma^+)) \to uco(\wp(\Sigma^+))$ that, given an abstract domain $\mu \in uco(\wp(\Sigma^+))$, returns the most concrete domain that abstracts μ and that is preserved by transformation t on program P. Formally:

$$K_{P,t} \stackrel{\text{def}}{=} \lambda\mu.\, \sqcap \left\{\ \varphi \in uco(\wp(\Sigma^+)) \,\middle|\, \mu \sqsubseteq \varphi \wedge \varphi(\mathbf{S}^+[\![P]\!]) = \varphi(t(\mathbf{S}^+[\![P]\!]))\ \right\}$$

Intuitively $K_{P,t}$ looses the minimal amount of information with respect to a given abstract domain in order to obtain a property preserved by t on P. Consequently, $K_{P,t}(id)$ is the most concrete property preserved by transformation t on program P. By definition $K_{P,t}(id)$ is a closure operator and it is therefore uniquely determined by the set of its fixpoints. In the following we characterize the elements of such closure in terms of a predicate on sets of traces.

Let us define $Pres_{P,t}(\mathcal{X})$ as a predicate over set of traces parametrized on a program P and a semantic transformation $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$ that, given a subset of program traces $\mathcal{X} \subseteq \mathbf{S}^+[\![P]\!]$, evaluates to *true* if and only if the set $\mathcal{X}$ is closed under transformation t, namely:

$$Pres_{P,t}(\mathcal{X}) = true \ \Leftrightarrow\ \forall \mathcal{Y} \subseteq \mathbf{S}^+[\![P]\!] : \mathcal{Y} \subseteq \mathcal{X} \Rightarrow t(\mathcal{Y}) \subseteq \mathcal{X}$$

Hence, the predicate $Pres_{P,t}(\mathcal{X})$ characterizes the set of traces $\mathcal{X} \in \wp(\Sigma^+)$ that are preserved by transformation t on program P. The following result shows how the collection of sets of traces $\mathcal{X} \in \wp(\Sigma^+)$ satisfying $Pres_{P,t}$ characterizes a semantic property preserved by transformation t on program P.

Lemma 4.6. Given a semantic transformation $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$ and a program $P \in \mathbb{P}$, the set $\varphi_{P,t}(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t}(\mathcal{X})\}$ is a closure, namely $\varphi_{P,t} \in \mathit{uco}(\wp(\Sigma^+))$. Moreover, property $\varphi_{P,t}$ is preserved by t on P.

PROOF: Let us show that $\{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t}(\mathcal{X})\}$ is a Moore family. It is clear that $\forall \mathcal{X} \subseteq \Sigma^+$: $t(\mathcal{X}) \subseteq \Sigma^+$, therefore Σ^+ is the top element and belongs to $\varphi_{P,t}(\wp(\Sigma^+))$. Moreover, $\{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t}(\mathcal{X})\}$ is closed under *glb*, namely given $\{\mathcal{X}_i\}_{i\in I}$ such that $\forall i \in I : \mathit{Pres}_{P,t}(\mathcal{X}_i) = \mathit{true}$, then $\mathit{Pres}(\cap_{i\in I}\mathcal{X}_i) = \mathit{true}$. In fact $\mathcal{Y} \subseteq \cap_{i\in I}\mathcal{X}_i$ means that $\forall i \in I : \mathcal{Y} \subseteq \mathcal{X}_i$, therefore by hypothesis $\forall i \in I : t(\mathcal{Y}) \subseteq \mathcal{X}_i$, meaning that $t(\mathcal{Y}) \subseteq \cap_{i\in I}\mathcal{X}_i$. Therefore, there exists a closure operator, denoted $\varphi_{P,t} \in \mathit{uco}(\wp(\Sigma^+))$, such that $\varphi_{P,t}(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t}(\mathcal{X})\}$. Let us show that $\varphi_{P,t}$ is preserved by t on P, namely that $\varphi_{P,t}(\mathbf{S}^+[\![P]\!]) = \varphi_{P,t}(t(\mathbf{S}^+[\![P]\!]))$. Assume that $\varphi_{P,t}(\mathbf{S}^+[\![P]\!]) \neq \varphi_{P,t}(t(\mathbf{S}^+[\![P]\!]))$, this means that there exist $\mathcal{X} \in \varphi_{P,t}$ such that $\mathbf{S}^+[\![P]\!] \subseteq \mathcal{X}$ while $t(\mathbf{S}^+[\![P]\!]) \not\subseteq \mathcal{X}$, which is impossible since $\mathcal{X} \in \varphi_{P,t}$ and therefore $\mathit{Pres}_{P,t}(\mathcal{X})$ holds.

□

Moreover, it is possible to show that the property $\varphi_{P,t}(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t}(\mathcal{X})\}$ induced by predicate $\mathit{Pres}_{P,t}$ is the most concrete property preserved by transformation t on program P.

Theorem 4.7. $K_{P,t}(id)(\wp(\Sigma^+)) = \left\{\ \mathcal{X} \in \wp(\Sigma^+) \,\middle|\, \mathit{Pres}_{P,t}(\mathcal{X})\ \right\}$.

PROOF: Let us show that $K_{P,t}(id) = \varphi_{P,t}$. By definition $K_{P,t}(id)$ is the most concrete preserved property, while from Lemma 4.6 $\varphi_{P,t}$ is a preserved property, therefore $K_{P,t}(id) \sqsubseteq \varphi_{P,t}$. We have to show that $\varphi_{P,t} \sqsubseteq K_{P,t}(id)$, namely $K_{P,t}(id)(\wp(\Sigma^+)) \subseteq \varphi_{P,t}(\wp(\Sigma^+))$. Let us assume that there exists an element $\mathcal{X} \in K_{P,t}(id)(\wp(\Sigma^+))$ such that $\mathcal{X} \notin \varphi_{P,t}(\wp(\Sigma^+))$. This means that $\mathit{Pres}_{P,t}(\mathcal{X}) = \mathit{false}$, namely that there exists $\mathcal{Y} \subseteq \mathbf{S}^+[\![P]\!]$ such that $\mathcal{Y} \subseteq \mathcal{X}$ while $t(\mathcal{Y}) \not\subseteq \mathcal{X}$, which implies $\mathcal{X} \notin K_{P,t}(id)(\wp(\Sigma^+))$, leading to a contradiction.

□

It follows that $K_{P,t}(id)(\wp(\Sigma^+)) = \{\mathcal{X} \in \wp(\Sigma^+) \mid \mathit{Pres}_{P,t}(\mathcal{X})\}$ is the most concrete property preserved by the transformation t on program P. Hence, the most concrete property preserved by t on all programs, is given by the least upper bound between the most concrete properties preserved on each program $P \in \mathbb{P}$ by t, i.e., $\bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)$. More precisely the following holds.

Theorem 4.8. Let $\mathbb{t} : \mathbb{P} \to \mathbb{P}$, then $\delta_{\mathbb{t}} = \bigsqcup_{P\in\mathbb{P}} K_{P,S^+\circ\mathbb{t}\circ\mathbb{P}}(id)$.

PROOF: Let us first show that $\bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)$ is the most concrete property preserved by transformation t on all programs. (1) $\bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)$ is preserved: observe that given a program $Q \in \mathbb{P}$ then $K_{Q,t}(id) \sqsubseteq \bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)$, by hypothesis $K_{Q,t}(id)$ is preserved by t on program Q, therefore $\forall Q \in \mathbb{P}$: $\bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)(S^+[\![Q]\!]) = \bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)(t(S^+[\![Q]\!]))$. (2) $\bigsqcup_{P\in\mathbb{P}} K_{P,t}(id)$ is the

most concrete property preserved by t. Consider $\eta \in uco(\wp(\Sigma^+))$ such that $\forall P \in \mathbb{P} : \eta(\mathbf{S}^+[\![P]\!]) = \eta(t(\mathbf{S}^+[\![P]\!]))$, then $\bigsqcup_{P\in\mathbb{P}} K_{P,t}(id) \sqsubseteq \eta$ iff $\forall P \in \mathbb{P} : K_{P,t}(id) \sqsubseteq \eta$ which is true since $K_{P,t}(id)$ is the most concrete property preserved by t on P. To conclude recall that t is an algorithmic transformation, therefore we can write $t = \mathbf{S}^+ \circ \mathbb{t} \circ \mathbb{p}$.

□

Example 4.9. Let us consider the semantic transformation $t : \wp(\Sigma^+) \to \wp(\Sigma^+)$ that given a set of traces $\mathcal{S} \in \wp(\Sigma^+)$, returns $t(\mathcal{S}) = \{t(\sigma) \mid \sigma \in \mathcal{S}\}$, where $t(\sigma) = t(\sigma_0, \ldots, \sigma_f) = \sigma_f$. Thus, given a program trace σ transformation t returns its final state σ_f. Therefore, $t(\mathbf{S}^+[\![P]\!])$ collects the final states of the execution of program P on every possible input. Given a program P, the set of traces that satisfy predicate $Pres_{P,t}$ corresponds to the set of traces that have the same final state. Formally for each final state σ_f of the execution of program P we define the set of traces ending with σ_f as $\mathcal{X}_{\sigma_f} = \{\mu \in \mathbf{S}^+[\![P]\!] \mid \mu_f = \sigma_f\}$. It is clear that for each $\mathcal{X}_{\sigma_f}$ we have that $Pres_{P,t}(\mathcal{X}_{\sigma_f})$ holds, in fact $\forall \mathcal{Y} \subseteq \mathbf{S}^+[\![P]\!]$: $\mathcal{Y} \subseteq \mathcal{X}_{\sigma_f} \Rightarrow t(\mathcal{Y}) \subseteq \mathcal{X}_{\sigma_f}$. Following Theorem 4.7 we have that $Final_P = \{\mathcal{X}_{\sigma_f} \mid \exists \sigma \in \mathbf{S}^+[\![P]\!] : \sigma = \sigma_0 \ldots \sigma_f\}$ is the most concrete property preserved by t on program P. This means that the most concrete property preserved by t on all programs is given by the least upper bound over all programs of the abstract domains $Final_P$, i.e., $\bigsqcup_{P\in\mathbb{P}} Final_P$, which is the closure that has as fixpoints the set of finite traces in Σ^+ that have the same final state.

□

4.2.2 Comparing Transformations

The semantics-based definition of obfuscation, that characterizes the obfuscating behaviour of a program transformation $\mathbb{t}$ in terms of the most concrete preserved property $\delta_{\mathbb{t}}$, allows us to compare obfuscating transformations with respect to their potency, namely according to the most concrete preserved property. In other words, it allows us to formalize a *partial order* relation between obfuscating transformations with respect to the sets of properties hidden by the transformations. On one hand, it is natural to think that a transformation is more potent than another one if it obfuscates larger amount of semantic properties. On the other hand, it may be interesting to compare the potency of different obfuscating transformations with respect to a particular property $\phi \in uco(\wp(\Sigma^+))$. In this second case, the idea is that a transformation $\mathbb{t}'$ is more potent than a transformation $\mathbb{t}$ with respect to ϕ if $\mathbb{t}'$ obfuscates property ϕ more than what $\mathbb{t}$ does. This means that $\mathbb{t}'$ is more efficient than $\mathbb{t}$ in hiding property ϕ of program semantics.

Definition 4.10. Given two program transformations $\mathbb{t}, \mathbb{t}' : \mathbb{P} \to \mathbb{P}$ and a semantic property $\phi \in uco(\wp(\Sigma^+))$ such that $\phi \in O_{\delta_{\mathbb{t}}} \cap O_{\delta_{\mathbb{t}'}}$, we have that:

- $\mathbb{t}'$ is *more potent* than $\mathbb{t}$, denoted by $\mathbb{t} \ll \mathbb{t}'$, if $O_{\delta_{\mathbb{t}}} \subseteq O_{\delta_{\mathbb{t}'}}$
- $\mathbb{t}'$ is *more potent* than $\mathbb{t}$ *with respect to* ϕ, denoted $\mathbb{t} \ll_\phi \mathbb{t}'$, if $\phi \ominus (\delta_{\mathbb{t}'} \sqcup \phi) \sqsubseteq \phi \ominus (\delta_{\mathbb{t}} \sqcup \phi)$

From the structure of the lattice of abstract interpretations $uco(\wp(\Sigma^+))$ it is possible to give an alternative characterization of the set $O_{\delta_{\mathbb{t}}}$ of properties obfuscated by a program transformation $\mathbb{t}$. This leads to the observation of some basic properties that relate transformations and preserved properties to the set of masked properties.

Proposition 4.11. Given two properties $\delta, \mu \in uco(\wp(\Sigma^+))$, we have that:

(1) $O_\delta = \{\mu \in uco(\wp(\Sigma^+)) \mid \mu \notin\uparrow \delta\}$
(2) If $\mu \sqsubset \delta$ then $O_\mu \subseteq O_\delta$, namely the transformation that preserves δ is more potent than the one that preserves μ
(3) $O_{\delta \sqcup \mu} = O_\delta \cup O_\mu$

PROOF:

(1) Recall that, given a lattice C and a domain D such that $C \sqsubseteq D$, then $C \ominus D = \top \Leftrightarrow C = D$ [67]. Thus: $\mu \ominus (\delta \sqcup \mu) \neq \top \;\Leftrightarrow\; \delta \sqcup \mu \neq \mu \;\Leftrightarrow\; \mu \notin\uparrow \delta$. Therefore, the set $\{\mu \in uco(\wp(\Sigma^+)) \mid \mu \ominus (\delta \sqcup \mu) \neq \top\}$ is equivalent to the set $\{\mu \in uco(\wp(\Sigma^+)) \mid \mu \notin\uparrow \delta\}$.
(2) We have to prove that $\forall \phi \in O_\mu$ then $\phi \in O_\delta$. By definition a property ϕ belongs to O_μ iff $\phi \in\uparrow \mu = \{\varphi \mid \mu \sqsubseteq \varphi\}$. By hypothesis $\mu \sqsubseteq \delta$, therefore if $\delta \sqsubseteq \varphi$ then $\mu \sqsubseteq \varphi$, therefore $\uparrow \delta \subseteq\uparrow \mu$. This means that if $\phi \notin\uparrow \mu$ then $\phi \notin\uparrow \delta$, namely if $\phi \in O_\mu$ then $\phi \in O_\delta$.
(3) We need to show that $\phi \notin\uparrow \delta \;\wedge\; \phi \notin\uparrow \mu \;\Leftrightarrow\; \phi \notin\uparrow (\delta \sqcup \mu)$. This is equivalent to $\phi \in\uparrow \delta \;\wedge\; \phi \in\uparrow \mu \;\Leftrightarrow\; \phi \in\uparrow (\delta \sqcup \mu)$, which is true since $\delta \sqsubseteq \phi \wedge \mu \sqsubseteq \phi \;\Leftrightarrow\; \delta \sqcup \mu \sqsubseteq \phi$.

□

4.3 Modeling Attackers

In the malicious reverse engineering setting an attacker is a malicious observer of the program behavior, whose task is to understand the inner workings of proprietary software systems in order to reuse the software for unlawful purposes or to make unauthorized modifications. The goal of code obfuscation is to make a program so difficult for an attacker to understand that reverse engineering it becomes uneconomical. Our semantics-based notion of code obfuscation given in Definition 4.4 characterizes an obfuscating transformation $\mathbb{t}$ in terms of the most concrete property it preserves. Hence, a δ-obfuscator $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is characterized by the most concrete property δ precisely observable on program semantics after

transformation $\mathbb{t}$. The idea is that what transformation $\mathbb{t}$ preserves is exactly what an attacker can still observe after obfuscation. Different attackers may be interested in different aspects of program behaviour, and they can be classified with respect to the precision of their observation. Thus, what an attacker deduces from the observation of an obfuscated program depends both on the property of interest of the attacker and on the particular obfuscation used.

Given the semantics-based definition of code obfuscation it comes natural to model attackers as abstract domains $\varphi \in uco(\wp(\Sigma^+))$. The idea is that an abstract domain expressing a certain property of program behaviours formally models the attacker interested in that property. The complete lattice of abstract domains $\langle uco(\wp(\Sigma^+)), \sqsubseteq \rangle$ provides here the right framework where to compare attackers with respect to their degree of abstraction and obfuscators with respect to their potency. On one hand the more concrete an attacker is, the bigger is the amount of information it needs to perform its intended damage on a program. On the other hand given a δ-obfuscator the more abstract δ is, the bigger is the potency of the obfuscating transformation. Our semantics-based approach, where code obfuscators are characterized by the most concrete preserved property and attackers are modeled as abstract domains, makes it possible to formally define the borderline between harmless and effective attackers with respect to a given obfuscation (i.e., preserved and obfuscated properties). Thus, we can say that a program transformation $\mathbb{t}$ is a $\delta_{\mathbb{t}}$-obfuscator, which is able to defeat all the attackers modeled by a property $\varphi \in uco(\wp(\Sigma^+))$ such that $\varphi \in O_{\delta_{\mathbb{t}}}$.

4.4 Case study: Constant Propagation

Constant propagation is a well-known program transformation that, knowing the variable values that are constant on all possible executions of a program, propagates these constant values as far forward through the program as possible. As discussed above, every program transformation can be viewed as a potential obfuscation by investigating the effects that such a transformation has on program semantics. In the following we are going to illustrate this idea clarifying the obfuscating behaviour of constant propagation.

Semantic aspects of Constant Propagation.

Let as recall how an efficient algorithm for constant propagation can be derived as an approximation of the corresponding semantic transformation [44].

Action Specialization

The *residual* $\mathbf{R}[\![D]\!]\rho$ of an arithmetic or boolean expression $D \in \mathbb{E} \cup \mathbb{B}$ in an environment ρ is the expression resulting from specializing D in such an environment (see Table 4.1). When expression D can be fully evaluated in environment

Arithmetic Expressions $\quad \mathbf{R} \in \mathbb{E} \times \mathfrak{E} \to \mathbb{E}$

$$
\begin{array}{rcl}
\mathbf{R}[\![n]\!]\rho & \stackrel{\text{def}}{=} & n \\
\mathbf{R}[\![X]\!]\rho & \stackrel{\text{def}}{=} & \text{if } X \in \mathit{dom}(\rho) \text{ then } \rho(X) \text{ else } X \\
\mathbf{R}[\![E_1 - E_2]\!]\rho & \stackrel{\text{def}}{=} & \text{let } E_1^r = \mathbf{R}[\![E_1]\!]\rho \text{ and } E_2^r = \mathbf{R}[\![E_2]\!]\rho \text{ in} \\
& & \quad \text{if } E_1^r = \mho \text{ or } E_2^r = \mho \text{ then } \mho \\
& & \quad \text{else if } E_1^r = n_1 \text{ and } E_2^r = n_2 \text{ then } n = n_1 - n_2 \\
& & \quad\quad \text{else } E_1^r - E_2^r
\end{array}
$$

Boolean Expressions $\quad \mathbf{R} \in \mathbb{B} \times \mathfrak{E} \to \mathbb{B}$

$$
\begin{array}{rcl}
\mathbf{R}[\![E_1 < E_2]\!]\rho & \stackrel{\text{def}}{=} & \text{let } E_1^r = \mathbf{R}[\![E_1]\!]\rho \text{ and } E_2^r = \mathbf{R}[\![E_2]\!]\rho \text{ in} \\
& & \quad \text{if } E_1^r = \mho \text{ or } E_2^r = \mho \text{ then } \mho \\
& & \quad \text{else if } E_1^r = n_1 \text{ and } E_2^r = n_2 \text{ and } b = n_1 < n_2 \text{ then } b \\
& & \quad\quad \text{else } E_1^r < E_2^r \\
\mathbf{R}[\![B_1 \vee B_2]\!]\rho & \stackrel{\text{def}}{=} & \text{let } B_1^r = \mathbf{R}[\![B_1]\!]\rho \text{ and } B_2^r = \mathbf{R}[\![B_2]\!]\rho \text{ in} \\
& & \quad \text{if } B_1^r = \mho \text{ or } B_2^r = \mho \text{ then } \mho \\
& & \quad \text{else if } B_1^r = \mathit{true} \text{ or } B_2^r = \mathit{true} \text{ then } \mathit{true} \\
& & \quad \text{else if } B_1^r = \mathit{false} \text{ then } B_2^r \\
& & \quad \text{else if } B_2^r = \mathit{false} \text{ then } B_1^r \\
& & \quad \text{else } B_1^r \vee B_2^r \\
\mathbf{R}[\![\neg B]\!]\rho & \stackrel{\text{def}}{=} & \text{let } B^r = \mathbf{R}[\![B]\!]\rho \text{ in} \\
& & \quad \text{if } B^r = \mho \text{ then } \mho \\
& & \quad \text{else if } B^r = \mathit{true} \text{ then } \mathit{false} \\
& & \quad \text{else if } B^r = \mathit{false} \text{ then } \mathit{true} \\
& & \quad \text{else } \neg B^r \\
\mathbf{R}[\![\texttt{true}]\!]\rho & \stackrel{\text{def}}{=} & \mathit{true} \\
\mathbf{R}[\![\texttt{false}]\!]\rho & \stackrel{\text{def}}{=} & \mathit{false}
\end{array}
$$

Table 4.1. Expression Specialization [44]

ρ, i.e., $\mathit{var}[\![D]\!] \subseteq \mathit{dom}(\rho)$, we say that expression D is *static* in the environment ρ, denoted $\mathbf{static}[\![D]\!]\rho$. When D is not static it is *dynamic*. It is clear that $\mathbf{static}[\![D]\!]\rho$ means that the specialization of expression D in environment ρ leads to a static value, i.e., a constant, $\mathbf{R}[\![D]\!]\rho \in \mathfrak{D}_\mho \cup \mathcal{B}_\mho$. Recall that the correctness of expression specialization follows from the fact that given two environments ρ and ρ' such that $\mathit{dom}(\rho) \subseteq \mathit{dom}(\rho')$ and $\forall x \in \mathit{dom}(\rho) : \rho(X) = \rho'(X)$, then $\mathbf{A}[\![\mathbf{R}[\![D]\!]\rho]\!]\rho' = \mathbf{A}[\![D]\!]\rho'$ and $\mathbf{A}[\![\mathbf{R}[\![D]\!]\rho]\!]\rho' = \mathbf{A}[\![\mathbf{R}[\![D]\!]\rho]\!](\rho' \setminus \mathit{dom}(\rho))$. The specialization of action A in environment ρ, denoted as $\mathbf{R}[\![A]\!]\rho$, produces both a residual action and a residual environment as defined in Table 4.2.

Let $\alpha^c_{\mathcal{O}}$ be the *observational abstraction* that has to be preserved by constant propagation in order to ensure the correctness of the transformation. In [44] abstraction $\alpha^c_{\mathcal{O}} : \wp(\Sigma^+) \to \wp(\mathfrak{E}^+)$ is defined as follows:

$$\alpha^c_{\mathcal{O}}(\mathcal{X}) \stackrel{\text{def}}{=} \left\{ \; \alpha^c_{\mathcal{O}}(\sigma) \,\middle|\, \sigma \in \mathcal{X} \; \right\} \quad \alpha^c_{\mathcal{O}}(\sigma) \stackrel{\text{def}}{=} \lambda i.\alpha^c_{\mathcal{O}}(\sigma_i) \quad \alpha^c_{\mathcal{O}}(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \rho$$

Actions	$\mathbf{R} \in \mathbb{A} \times \mathfrak{E} \to \mathfrak{E} \times \mathbb{A}$
$\mathbf{R}[\![B]\!]\rho$	$\stackrel{\text{def}}{=} \langle \rho, \mathbf{R}[\![B]\!]\rho \rangle$
$\mathbf{R}[\![X :=?]\!]\rho$	$\stackrel{\text{def}}{=} \langle \rho \setminus X, X :=? \rangle$
$\mathbf{R}[\![X := E]\!]\rho$	$\stackrel{\text{def}}{=}$ if $\mathbf{static}[\![E]\!]\rho$ then $\langle \rho[X := \mathbf{R}[\![E]\!]\rho], \texttt{skip} \rangle$
	else $\langle \rho \setminus X, X := \mathbf{R}[\![E]\!]\rho \rangle$

Table 4.2. Action Specialization [44]

Thus, function $\alpha^c_{\mathcal{O}}$ abstracts from the particular commands that produce a certain environment evolution keeping only the environment trace.

Given a set of traces $\mathcal{X} \in \wp(\Sigma^+)$, let $\mathcal{X}^c$ denote the result of a preliminary static analysis detecting constants. Formally $\mathcal{X}^c$ is a sound approximation of $\alpha^c(\mathcal{X})$ where:

$$\alpha^c(\mathcal{X}) = \lambda L.\lambda X. \dot{\bigsqcup} \left\{ \rho(X) \,\middle|\, \exists \sigma \in \mathcal{X} : \exists C \in \mathbb{C} : \exists i : \sigma_i = \langle \rho, C \rangle, lab[\![C]\!] = L \right\}$$

where $\dot{\bigsqcup}$ is the pointwise extension of the least upper bound $\bigsqcup$ in the complete lattice $\mathfrak{D}^c \stackrel{\text{def}}{=} \mathfrak{D}_{\mathcal{U}} \cup \{\top, \bot\}$, where $\forall x \in \mathfrak{D}^c : \bot \sqsubseteq x \sqsubseteq x \sqsubseteq \top$. This means that, given a program P and a label $L \in lab[\![P]\!]$, $\alpha^c(\mathbf{S}^+[\![P]\!])(L)$ is an environment mapping (denoted ρ^c_L for short when the set of traces is clear from the context) that, given a variable $X \in var[\![P]\!]$, returns the value of X if X is constant at program point L, $\top$ otherwise. Thus a variable X of program P has a constant value at program point L when $\alpha^c(\mathbf{S}^+[\![P]\!])(L)(X) \neq \top$, i.e., $\rho^c_L(X) \neq \top$. The semantic transformation $t^c : \wp(\Sigma^+) \times \alpha^c(\wp(\Sigma^+)) \to \wp(\Sigma^+)$ performing constant propagation is constructively defined as follows:

$$t^c[\mathcal{X}, \mathcal{X}^c] \stackrel{\text{def}}{=} \left\{ t^c[\sigma, \mathcal{X}^c] \,\middle|\, \sigma \in \mathcal{X} \right\}$$

$$t^c[\sigma, \mathcal{X}^c] \stackrel{\text{def}}{=} \lambda i. t^c[\sigma_i, \mathcal{X}^c] \quad t^c[\langle \rho, C \rangle, \mathcal{X}^c(lab[\![C]\!])] \stackrel{\text{def}}{=} \langle \rho, t^c[C, \rho^c_{lab[\![C]\!]}] \rangle$$

where command specialization is defined as:

$$t^c[L : A \to L', \rho^c_L] \stackrel{\text{def}}{=} L : t^c[A, \rho^c_L] \to L'$$

$$t^c[A, \rho^c_L] = \text{ let } \langle \rho_r, A_r \rangle \stackrel{\text{def}}{=} \mathbf{R}[\![A]\!]\rho|_{\{X \in \mathbb{X} | \rho^c_L(X) \in \mathfrak{D}_{\mathcal{U}}\}} \text{ in } A_r$$

The correctness of t^c follows from the fact that the transformed traces are valid traces, i.e., $\sigma \in \Sigma^+ \Rightarrow t^c[\sigma, \mathcal{X}^c] \in \Sigma^+$, and that $\alpha^c_{\mathcal{O}}$ is preserved by t^c since the transformation leaves the environments unchanged [44].

Example 4.12. Let us consider the program in Table 4.3 and its execution trace $\sigma = \sigma_1\sigma_2\sigma_3\sigma_4....$ Let us represent the state environment of this program as a tuple $(v_a, v_b, v_c, v_d, v_e)$ of values corresponding to the variables a, b, c, d, e, and let us assume that condition B holds *true* in state σ_2. Then the states of trace σ are given by:

```
a:= 1; b:=2; c:=3; d:=3; e:=0;
while B do
    b:=2*a; d:=d+1; e:=e-a;
    a:=b-a; c:=e+d;
endw
```

L_1 : `a:= 1; b:=2; c:=3; d:=3; e:=0;` $\rightarrow L_2$
L_2 : `B` $\rightarrow L_3$
L_2 : $\neg$`B` $\rightarrow L_5$
L_3 : `b:=2*a; d:=d+1; e:=e-a;` $\rightarrow L_4$
L_4 : `a:=b-a; c:=e+d;` $\rightarrow L_2$
L_5 : `stop` $\rightarrow \not{l}$

Table 4.3. A simple program from [38]

- $\sigma_1 = \langle(\bot, \bot, \bot, \bot, \bot), L_1 : \texttt{a:= 1; b:=2; c:=3; d:=3; e:=0;} \rightarrow L_2\rangle$
- $\sigma_2 = \langle(1, 2, 3, 3, 0), L_2 : B \rightarrow L_3\rangle$
- $\sigma_3 = \langle(1, 2, 3, 3, 0), L_3 : \texttt{b:=2*a; d:=d+1; e:=e-a;} \rightarrow L_4\rangle$
- $\sigma_4 = \langle(1, 2, 3, 4, -1), L_4 : \texttt{a:=b-a; c:=e+d;} \rightarrow L_2\rangle$
- $\sigma_5 = ...$

In this case the preliminary static analysis $\mathcal{X}^c$ observes that variables a and b are actually constants at labels L_2, L_3 and L_4. Therefore, following the above definitions, the transformed trace $t^c[\sigma, \mathcal{X}^c]$ is given by the following sequence of transformed states:

- $t^c[\sigma_1, \mathcal{X}^c(L_1)] = \sigma_1$
- $t^c[\sigma_2, \mathcal{X}^c(L_2)] = \sigma_2$
- $t^c[\sigma_3, \mathcal{X}^c(L_3)] = \langle(1, 2, 3, 3, 0), L_3 : \texttt{skip; d:=d+1; e:=e-a;} \rightarrow L_4\rangle$
- $t^c[\sigma_4, \mathcal{X}^c(L_4)] = \langle(1, 2, 3, 4, -1), L_4 : \texttt{skip; c:=e+d;} \rightarrow L_2\rangle$
- $t^c[\sigma_5, \mathcal{X}^c(L) = ...$

We can observe that transformation t^c, knowing that variables a and b are constants, modifies the states σ_3 and σ_4 where assignments to a and b are replaced with skip actions.

□

Following the steps elucidated at the end of Section 2.3 it is possible to derive a constant propagation algorithm $\mathbb{t}^c = \mathbb{p} \circ t^c \circ \mathbf{S}^+$. We omit here such details because they are not significant for our reasoning.

Obfuscating Behaviour of Constant Propagation.

In order to understand the obfuscating behaviour of constant propagation we need to consider the most concrete property δ_{t^c} preserved by the previously defined transformation t^c. Following the characterization proposed by Theorem 4.8 we can formalize δ_{t^c} as follows:

$$\delta_{t^c} = \bigsqcup_{P \in \mathbb{P}} \left\{ \mathcal{X} \in \wp(\Sigma^+) \,\middle|\, \mathit{Pres}_{P,t^c}(\mathcal{X}) \right\}$$

Where, given a set of traces $\mathcal{X} \in \wp(\Sigma^+)$:

$$\begin{aligned} &\mathit{Pres}_{P,t^c}(\mathcal{X}) = \mathit{true} \;\Leftrightarrow \\ &\qquad \forall \mathcal{Y} \subseteq \mathbf{S}^+[\![P]\!] : \mathcal{Y} \subseteq \mathcal{X} \Rightarrow \forall \mathbf{S}^c[\![P]\!] \sqsupseteq \alpha^c(\mathbf{S}^+[\![P]\!]) : t^c[\mathcal{Y}, \mathbf{S}^c[\![P]\!]] \subseteq \mathcal{X} \end{aligned}$$

This means that a set of traces $\mathcal{X}$ is a fixpoint of closure δ_{t^c} if it contains the specialization of its traces according to any sound constant analysis. Namely for each trace σ in $\mathcal{X}$ it holds that: $\{\eta = t^c[\sigma, \mathbf{S}^c[\![P]\!]] \mid \alpha^c(\mathbf{S}^+[\![P]\!]) \sqsubseteq \mathbf{S}^c[\![P]\!]\} \subseteq \mathcal{X}$.

Let $\varphi^c_{\mathcal{O}} = \gamma^c_{\mathcal{O}} \circ \alpha^c_{\mathcal{O}}$ be the closure operator corresponding to the observational abstraction $\alpha^c_{\mathcal{O}}$, where $\gamma^c_{\mathcal{O}}$ is the concretization map induced by abstraction $\alpha^c_{\mathcal{O}}$. It is clear that, since the observational abstraction $\alpha^c_{\mathcal{O}}$ is preserved by t^c, then $\varphi^c_{\mathcal{O}} \in \mathit{uco}(\wp(\Sigma^+))$ is preserved by transformation t^c. This means that, by definition of δ_{t^c}, we have $\delta_{t^c} \sqsubseteq \varphi^c_{\mathcal{O}}$ and therefore $\varphi^c_{\mathcal{O}} \ominus (\varphi^c_{\mathcal{O}} \sqcup \delta_{t^c}) = \top$, which, from a code obfuscation point of view, means that the attacker modeled by property $\varphi^c_{\mathcal{O}}$ is not obfuscated by constant propagation transformation.

On the other hand, let us consider property $\theta = \gamma_\theta \circ \alpha_\theta \in \mathit{uco}(\wp(\Sigma^+))$, observing the successions of environments and types of actions, namely:

$$\alpha_\theta(\mathcal{X}) \stackrel{\text{def}}{=} \left\{ \alpha_\theta(\sigma) \,\middle|\, \sigma \in \mathcal{X} \right\} \qquad \alpha_\theta(\sigma) \stackrel{\text{def}}{=} \lambda i.\alpha_\theta(\sigma_i)$$

$$\alpha_\theta(\langle \rho, C \rangle) \stackrel{\text{def}}{=} (\rho, \mathit{type}[\![\mathit{act}[\![C]\!]]\!])$$

where *type* maps actions into the following set of action types $\{\mathit{assign}, \mathit{skip}, \mathit{test}\}$. It is clear that this property is not preserved by t^c, since, in general $\mathit{type}[\![A]\!] \neq \mathit{type}[\![\mathbf{R}[\![A]\!]\rho]\!]$ (see Example 4.13). This means that property θ is obfuscated by constant propagation, namely $\theta \in O_{\delta_{t^c}}$, i.e., $\theta \ominus (\theta \sqcup \delta_{t^c}) \neq \top$. By definition it follows that $O_{\delta_{t^c}} \neq \varnothing$ and therefore t^c is a δ_{t^c}-obfuscator according to Definition 4.4.

Example 4.13. As observed above, θ is not preserved by t^c, namely it could happen that: $\theta(\mathbf{S}[\![P]\!]) \neq \theta(t^c[\mathbf{S}^+[\![P]\!], \mathbf{S}^c[\![P]\!]])$. Once again let us consider the program in Table 4.3. In particular, we focus on the states that are modified by the transformation t^c, namely on:

- $\sigma_3 = \langle (1,2,3,3,0), L_3 : \texttt{b:=2*a; d:=d+1; e:=e-a;} \rightarrow L_4 \rangle$
- $\sigma_4 = \langle (1,2,3,4,-1), L_4 : \texttt{a:=b-a; c:=e+d;} \rightarrow L_2 \rangle$

recall that their transformed versions are respectively given by:

- $t^c[\sigma_3, \mathcal{X}^c(L_3)] = \langle (1,2,3,3,0), L_3 : \texttt{skip; d:=d+1; e:=e-a;} \rightarrow L_4 \rangle$
- $t^c[\sigma_4, \mathcal{X}^c(L_4)] = \langle (1,2,3,4,-1), L_4 : \texttt{skip; c:=e+d;} \rightarrow L_2 \rangle$

In this case, property θ on the original states observes:

– $\theta(\sigma_3) = \langle (1,2,3,3,0), L_3, L_4, \mathit{assign}, \mathit{assign}, \mathit{assign}\rangle$
– $\theta(\sigma_4) = \langle (1,2,3,4,-1), L_4, L_2, \mathit{assign}, \mathit{assign}\rangle$

while on the transformed states observes:

– $\theta(t^c[\sigma_3, \mathcal{X}^c(L_3)]) = \langle (1,2,3,3,0), L_3,\ L_4, \mathit{skip}, \mathit{assign}, \mathit{assign}\rangle$
– $\theta(t^c[\sigma_4, \mathcal{X}^c(L_4)]) = \langle (1,2,3,4,-1), L_4, L_2,\ \mathit{skip}, \mathit{assign}\rangle$

showing that the property θ is not preserved.

□

Moreover, we can show that what transformation t^c hides of property θ is the type of actions. In fact, consider the closure $\eta \in \mathit{uco}(\wp(\Sigma^+))$ which observes the *type* of actions:

$$\eta = \lambda\mathcal{X}.\ \left\{\ \sigma \ \middle|\ \begin{array}{l}\sigma' \in \mathcal{X} \text{ and } \forall i.\ \sigma_i = \langle \rho_i, C_i\rangle, \sigma'_i = \langle \rho'_i, C'_i\rangle \\ \mathit{type}(C_i) = \mathit{type}(C'_i)\end{array}\ \right\}$$

Theorem 4.14. $\theta \ominus (\theta \sqcup \delta_{t^c}) = \eta$.

PROOF: Let us prove that $\theta \sqcup \delta_{t^c} = \varphi^c_{\mathcal{O}}$. By definition of δ_{t^c} it follows that $\delta_{t^c} \sqsubseteq \varphi^c_{\mathcal{O}}$. Let us show that $\theta \sqsubseteq \varphi^c_{\mathcal{O}}$, namely that $\theta(\wp(\Sigma^+)) \subseteq \varphi^c_{\mathcal{O}}(\wp(\Sigma^+))$.

$$\theta(\mathcal{X}) = \left\{\ \sigma \ \middle|\ \begin{array}{l}\sigma' \in \mathcal{X} \text{ and } \forall i.\ \sigma_i = \langle \rho_i, C_i\rangle, \sigma'_i = \langle \rho_i, C'_i\rangle \\ \mathit{type}(C_i) = \mathit{type}(C'_i)\end{array}\ \right\}$$

$$\varphi^c_{\mathcal{O}}(\mathcal{X}) = \left\{\ \sigma \ \middle|\ \sigma' \in \mathcal{X} \text{ and } \forall i.\ \sigma_i = \langle \rho_i, C_i\rangle, \sigma'_i = \langle \rho_i, C'_i\rangle\ \right\}$$

Thus $\forall \mathcal{X} \in \wp(\Sigma^+) : \theta(\mathcal{X}) \subseteq \varphi^c(\mathcal{X})$ and therefore $\theta \sqsubseteq \varphi^c_{\mathcal{O}}$. Moreover $\varphi^c_{\mathcal{O}}$ is the most concrete property that θ and δ_{t^c} have in common. In fact it is clear that $\theta = \varphi^c_{\mathcal{O}} \sqcap \eta$, and since the *type* of actions, i.e., η, is not preserved by t^c we have that θ and δ_{t^c} share only the observation of the environments. Hence, we have that $\theta \ominus (\theta \sqcup \delta_{t^c}) = \theta \ominus \varphi^c_{\mathcal{O}} = (\varphi^c_{\mathcal{O}} \sqcap \eta) \ominus \varphi^c_{\mathcal{O}} = \eta$. Where the last equation holds since η is the most abstract domain which reduced product with $\varphi^c_{\mathcal{O}}$ returns θ.

□

This means that constant propagation acts as an obfuscating transformation that defeats, for example, the attacker modeled by the abstract domain θ, while it is harmless with respect to the attacker modeled by property $\varphi^c_{\mathcal{O}}$.

4.5 Discussion

In this chapter we have introduced a generalized notion of code obfuscation, where a program transformation can be seen as an obfuscation even if it does

not preserve the observational behaviour of programs, i.e., their denotational semantics. In fact, following our definition, any program transformation can be seen as a potential obfuscator. The point here is that a transformation behaves as an obfuscator if there exists an attacker, i.e., a semantic property, that the transformation obstructs. For example, in order to defeat an attacker that is interested in something weaker than the input-output behaviour of a program, namely in something that can be deduced by program denotational semantics, we need an obfuscator that preserves less than denotational semantics, namely an obfuscator that masks something of the input-output behaviour of a program.

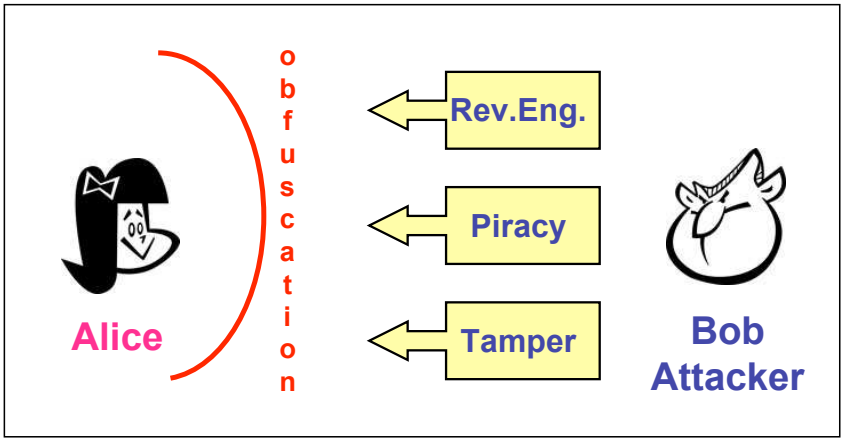
In the proposed framework, obfuscating transformations and attackers are both characterized by abstract domains. It is clear that being able to tune the most concrete property that a transformation preserves would allow us to modify the class of attackers that the transformation defeats. An interesting research task considers the possibility of using a systematic methodology for deriving program transformations in order to design obfuscating algorithms that are able to mask a desired property, namely to defeat a given attacker. Given an abstract domain modeling the most powerful attacker in a certain scenario, the idea is to derive the "simplest" transformation that protects a given class of programs against such an attacker.

If on the one hand, the generality of our definition as been proved by studying the obfuscating behaviour of constant propagation, on the other hand, the semantics-based definition turns out to be very useful in understanding the behaviour and potency of commonly used obfuscating transformations. This is shown in Chapter 5, where we consider the widely used obfuscation performing opaque predicate insertion. In particular, the semantic understanding of opaque predicate insertion, together with the idea of modeling attackers as abstract domains, allows us to characterize the ability of an attacker to reverse opaque predicate insertion as a completeness problem in the abstract interpretation sense. In Chapter 5 we will discuss how this result may lead to significant improvements in the performance of opaque predicate detection algorithms.

Recall that malware writers (i.e., hackers) often use code obfuscation techniques to prevent detection. This means that, when hackers use a δ-obfuscator, they obtain different malware versions that are semantically equivalent up to abstraction δ. Following this observation, in Chapter 6 we develop a theoretical framework for malware detectors based on program semantics and abstract interpretation, where program infection is specified as a matching relation between the (abstract) semantics of the malware and the (abstract) semantics of the program.

5

Control Code Obfuscation



The Malicious Host Perspective

In this chapter, we focus our attention on the semantic understanding of an interesting and widely used class of obfuscating transformations known as control code transformations. In particular, we consider control code obfuscation by opaque predicate insertion which adds fake conditional branches that may confuse the control flow of the original program. The idea is that an attacker that is not aware of the always constant value of an opaque predicate has to consider both branches (even if one is never executed at run time). In Section 5.1.1 we define the semantic transformation that formalizes the effects of opaque predicate insertion on program trace semantics, and then, in Section 5.1.2, we derive the corresponding obfuscating algorithm following the methodology proposed by Cousot and Cousot in [44]. The programming language considered in this chapter is the one described in Section 2.3. In Section 5.1.3 we observe that,

in the case of opaque predicate insertion, the obfuscating transformation has minor effects on concrete program semantics. In fact, every time that the concrete semantics evaluates an opaque predicate this returns a constant value, meaning that the execution always follows the same branch. Something different happens if we consider the abstract semantics computed by an attacker on the abstract domain modeling it as discussed in Section 5.1.4. As observed in Section 4.3, attackers are modeled as abstract domains, where the abstraction encodes the level of precision of the attacker in observing program behaviour. It turns out that an attacker is able to break opaque predicate insertion only if its abstraction is precise enough to detect the inserted opaque predicates. In Section 5.2 we briefly present some standard opaque predicate detection algorithms and their major drawbacks. Then, in Section 5.3 we consider a particular class of commonly used numerical opaque predicates for which the degree of precision needed to disclose opaqueness can be formalized as a completeness problem in the abstract interpretation field. In fact, in this case, the standard notion of complete domain precisely captures the amount of information needed by an attacker to disclose an opaque predicate. Based on this theoretical result, in Section 5.3.2, we propose a methodology, based on program semantics and abstract interpretation, to detect and then eliminate opaque predicates. Experimental evaluations show the efficiency of this detection algorithm. It is clear that the proposed abstract approach can be extended to other classes of opaque predicates. As an example, in Section 5.3.3 we consider another family of numerical opaque predicates characterized by a common structure, and also in this case the problem of opaque predicate detection can be reduced to a completeness problem of the abstract domain modeling the attacker. To conclude, we present some interesting research tasks that we plan to address in the future. The results presented in this chapter have been published in [47, 49].

5.1 Control Code Obfuscation

With control code obfuscators we refer to obfuscating techniques that act by masking the control flow behaviour of the original program. These transformations are often based on the insertion of opaque predicates. Following a standard definition, a predicate is opaque if its value is known a priori to the obfuscation, but this value is difficult for a deobfuscator to deduce [35]. In this chapter we refer to two major types of opaque predicates presented in Section 3.1.2: true opaque predicates P^T that always evaluate to *true* and false opaque predicates P^F that always evaluate to *false*. Given such constructs, it is possible to design transformations that break up the flow of control of programs by adding branch instructions controlled by opaque predicates and inserting dead or buggy code in the never executed path. In the following we focus on the insertion of true

opaque predicates, but analogous results can be obtained for false opaque predicates as well. In particular, when inserting a branch instruction controlled by an opaque predicate P^T, the *true* path starts with the next action of the original program, while the *false* path leads to termination or buggy code. This confuses the attacker who is not aware of the always true value of the opaque predicate, and he/she has to consider both paths. It is clear that this transformation does not heavily affect program semantics, since at run time the opaque predicate is always evaluated *true* and the true path is the only one to be executed. In fact, opaque predicate insertion aims at confusing the program control flow and this may not have major effects on program trace semantics (recall that control flow is an abstraction of trace semantics).

In the following we define the semantic transformation $t^{OP} : \wp(\Sigma^+) \to \wp(\Sigma^+)$ that mimics the effects of opaque predicate insertion on program trace semantics. In particular, t^{OP} transforms the semantics of the original program by simply adding opaque tests, which clearly modifies the structure of traces. Following the methodology proposed in [44] and elucidated in Section 2.3, we derive from t^{OP} the corresponding syntactic transformation $\mathbb{p}^+ \circ t^{OP} \circ \mathbf{S}^+$. The so obtained syntactic transformation is then extended to $\mathbb{t}^{OP}$ that performs opaque predicate insertion. In fact, the syntactic transformation $\mathbb{t}^{OP}$ inserts true opaque predicates (as $\mathbb{p}^+ \circ t^{OP} \circ \mathbf{S}^+$) together with their potential false paths (added manually to $\mathbb{p}^+ \circ t^{OP} \circ \mathbf{S}^+$). Next, we study the obfuscating behaviour of opaque predicate insertion with respect to attackers modeled, as usual, by abstract domains.

5.1.1 Semantic Opaque Predicate Insertion

Let $\mathfrak{I} : \mathbb{P} \to \wp(\mathbb{L})$ be the result of a preliminary static analysis that given a program returns the subset of its labels, i.e., program points, where it is possible to insert opaque predicates. Usually the preliminary static analysis consists of a combination of liveness analysis and static analyses. On the one hand, liveness analysis is typically used to ensure that no dependencies are broken by the inserted predicate and that the obfuscated program is functionally equivalent to the original one. On the other hand, static analyses, such as constant propagation, may be used to check whether opaque predicates have definite values *true* or *false*, namely if the predicate can be trivially broken. Given a program P, we assume to know the set $\mathfrak{I}[\![P]\!] \subseteq \mathit{lab}[\![P]\!]$ of labels that the preliminary static analysis has classified as candidate for opaque predicate insertion.

Given a set OP of true opaque predicates, let $\mathcal{X} \in \wp(\Sigma^+)$ be a set of traces, $\mathbb{K} \in \wp(\mathbb{L})$ be a set of labels, $P^T \in OP$ be a true opaque predicate and $\tilde{L}$ be an unused memory location, i.e., $\tilde{L} \notin \mathit{lab}[\![\mathbb{p}^+(\mathcal{X})]\!]$. The *semantic opaque predicate insertion* transformation $t^{OP} : \wp(\Sigma^+) \times \wp(\mathbb{L}) \to \wp(\Sigma^+)$ is defined as follows:

$$t^{OP}[\mathcal{X}, \mathbb{K}] \stackrel{\text{def}}{=} \{\ t^{OP}[\sigma, \mathbb{K}] \,|\, \sigma \in \mathcal{X}\ \}$$

$$t^{OP}[\langle \rho, L : A \to L' \rangle \sigma, \mathbb{K}] \stackrel{\text{def}}{=}$$

$$\begin{cases} \langle \rho, L : A \to L' \rangle \; t^{OP}[\sigma, \mathbb{K}] & \text{if } L \notin \mathbb{K} \\ \langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \; t^{OP}[\sigma, \mathbb{K}] & \text{if } L \in \mathbb{K} \end{cases}$$

By definition, transformation t^{OP} changes each trace of $\mathcal{X}$ independently and state by state. In particular, let L be a candidate label for opaque predicate insertion, and let $\langle \rho, L : A \to L' \rangle$ be the (original) program state which command is labelled by L. Transformation t^{OP} inserts the opaque predicate P^T at the candidate label L with co-label $\tilde{L}$, obtaining the transformed state $\langle \rho, L : P^T \to \tilde{L} \rangle$. To preserve program functionality, action A has to be the first action of the true branch of the opaque predicate P^T. This is guaranteed by inserting the new state $\langle \rho, \tilde{L} : A \to L' \rangle$. Thus, transformation t^{OP} preforms opaque predicate insertion by replacing state $\langle \rho, L : A \to L' \rangle$ with the two states $\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle$. It is clear that program environment remains unchanged since test actions, such as opaque predicates, don't affect the values of variables (at least in our model). Fig. 5.1 shows how program traces are modified by opaque predicate insertion. It is clear that the semantic transformation t^{OP},

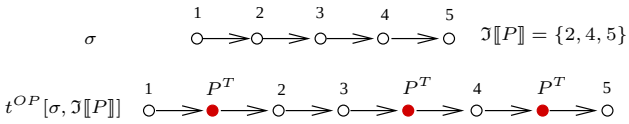


Fig. 5.1. Semantic opaque predicate insertion

that transforms traces by inserting opaque predicates from OP in the allowed program points ($\in \mathbb{K}$), transforms finite traces into finite traces.

Lemma 5.1. Given $\sigma \in \Sigma^+$ and $\mathbb{K} \in \wp(\mathbb{L})$, then $t^{OP}[\sigma, \mathbb{K}] \in \Sigma^+$.

PROOF: Given $\sigma \in \Sigma^+$, let $|\sigma| = n$ and $\forall i : 0 \leq i \leq n$ let $\sigma_i = \langle \rho_i, L_i : A_i \to L_{i+1} \rangle$. Observe that $\forall i \in [1, n-1]$ the transformation of the subtrace $\sigma_{i-1}\sigma_i\sigma_{i+1}$ of σ is still a trace, namely $\sigma'_{i-1}t^{OP}[\sigma_i\sigma_{i+1}, \mathbb{K}] \in \Sigma^+$. Two are the cases that we have to consider. (1) If $L_i \notin \mathbb{K}$, then we have that $\sigma'_{i-1}t^{OP}[\sigma_i\sigma_{i+1}, \mathbb{K}] = \sigma'_{i-1}\sigma_i\sigma'_{i+1}$, and $\sigma_i \in \mathbf{C}(\sigma'_{i-1})$ and $\sigma'_{i+1} \in \mathbf{C}(\sigma_i)$ follows form $\sigma \in \Sigma^+$. (2) On the other hand, if $L_i \in \mathbb{O}$ we have that:

$$\begin{aligned} \sigma'_{i-1}t^{OP}[\sigma_i\sigma_{i+1}, \mathbb{K}] &= \sigma'_{i-1}\langle \rho_i, L_i : P^T \to \tilde{L}_i \rangle \langle \rho_i, \tilde{L}_i : A_i \to L_{i+1} \rangle \sigma'_{i+1} \\ &= \sigma'_{i-1}\sigma^a_i\sigma^b_i\sigma'_{i+1} \end{aligned}$$

where $\sigma^a_i = \langle \rho_i, L_i : P^T \to \tilde{L}_i \rangle$ and $\sigma^b_i = \langle \rho_i, \tilde{L}_i : A_i \to L_{i+1} \rangle$. The test action given by the opaque predicate does not change the state environment and it

is clear that $\sigma_i^a \in \mathbf{C}(\sigma'_{i-1})$, $\sigma_i^b \in \mathbf{C}(\sigma_i^a)$ and $\sigma'_{i+1} \in \mathbf{C}(\sigma_i^b)$. This holds also for the initial and final state, in fact if $L_0 \in \mathbb{K}$ then $\sigma_1 \in \mathbf{C}(\langle \rho_0, L_0 : \tilde{A_0} \rightarrow L_1 \rangle)$ and if $L_n \in \mathbb{K}$ then $\langle \rho_n, L_n : P^T \rightarrow \tilde{L_n} \rangle \in \mathbf{C}(\sigma_{n-1})$. This proves that given $\eta = t^{OP}[\sigma, \mathbb{K}]$ then $\forall i$: $\eta_i \in \mathbf{C}(\eta_{i-1})$. Moreover if $|\mathbb{K}| = h$ then $|\eta| = n + h = k$, thus $\eta \in \Sigma^+$.

□

5.1.2 Syntactic Opaque Predicate Insertion

Given the semantic transformation t^{OP} it is possible, following the procedure elucidated in Section 2.3, to derive the syntactic transformation performing opaque predicate insertion. In particular, transformation $\mathbb{p}^+ \circ t^{OP} \circ \mathbf{S}^+$ simply inserts in a program commands whose actions are true predicates from OP. Such syntactic transformations can be easily extended to perform code obfuscation based on opaque predicate insertion (denoted as $\mathbb{t}^{OP}$ in the following), by inserting in the transformed program also the dead code following the false branch of P^T. In fact, following the definition of $\mathbb{p}^+$, these instructions cannot be present in $\mathbb{p}^+ \circ t^{OP} \circ \mathbf{S}^+$, since the commands of the never executed false path are not present in the transformed program semantics.

Following the methodology proposed by Cousot and Cousot [44], we systematically derive the algorithm performing opaque predicate insertion from its semantic counterpart t^{OP}.

Step 1: When considering program semantics in fixpoint form, the syntactic transformation $\mathbb{p}^+(t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]])$, reduces to $\mathbb{p}^+(t^{OP}[lfpF^+[\![P]\!], \mathfrak{I}[\![P]\!]])$.

Step 2: Let us compute the transformation t^{OP} of program semantics $\mathbf{S}^+[\![P]\!]$ expressed in fixpoint form $lfpF^+[\![P]\!]$, in order to establish the local commutation property necessary for fixpoint transfer:

$$t^{OP}[F^+[\![P]\!](\mathcal{X}), \mathfrak{I}[\![P]\!]] = t^{OP}[\mathfrak{T}[\![P]\!] \cup \{ss'\sigma \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathcal{X}\}, \mathfrak{I}[\![P]\!]] = t^{OP}[\mathfrak{T}[\![P]\!], \mathfrak{I}[\![P]\!]] \cup t^{OP}[\{ss'\sigma \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathcal{X}\}, \mathfrak{I}[\![P]\!]]$$

Let us consider the two terms of the above union separately. For the first term we have:

$$t^{OP}[\mathfrak{T}[\![P]\!], \mathfrak{I}[\![P]\!]] = \{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in \mathfrak{T}[\![P]\!]\} =$$

$$\{t^{OP}[\langle \rho, L : A \rightarrow L' \rangle, \mathfrak{I}[\![P]\!]] \mid L : A \rightarrow L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathcal{L}[\![P]\!]\} =$$

$$\{\langle \rho, L : A \rightarrow L' \rangle \mid L : A \rightarrow L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathcal{L}[\![P]\!], L \notin \mathfrak{I}[\![P]\!]\} \ \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L}\rangle\langle \rho, \tilde{L} : A \to L'\rangle | L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathcal{L}[\![P]\!], \\ L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

Considering the second term, we have that:

$$t^{OP}[\{ss'\sigma \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathcal{X}\}, \mathfrak{I}[\![P]\!]] = \\ \{t^{OP}[ss'\sigma, \mathfrak{I}[\![P]\!]] \mid s' \in \mathbf{C}[\![P]\!](s), s'\sigma \in \mathcal{X}\}$$

assuming $s = \langle \rho, L : A \to L'\rangle, s' = \langle \rho', C'\rangle$, we obtain:

$$\{\langle \rho, L : A \to L'\rangle t^{OP}[\langle \rho', C'\rangle\sigma, \mathfrak{I}[\![P]\!]] \mid lab[\![C']\!] = L', \rho' \in \mathbf{A}[\![A]\!]\rho, \\ L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \langle \rho', C'\rangle\sigma \in \mathcal{X}, L \notin \mathfrak{I}[\![P]\!]\} \quad \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L}\rangle\langle \rho, \tilde{L} : A \to L'\rangle t^{OP}[\langle \rho', C'\rangle\sigma, \mathfrak{I}[\![P]\!]] \mid lab[\![C']\!] = L', \\ \rho' \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \langle \rho', C'\rangle\sigma \in \mathcal{X}, L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

that, given $\sigma' = \langle \rho', C'\rangle\sigma$, reduces to:

$$\{\langle \rho, L : A \to L'\rangle t^{OP}[\sigma', \mathfrak{I}[\![P]\!]] \mid lab[\sigma'] = L', env[\sigma'] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \\ \rho \in \mathfrak{E}[\![P]\!], \sigma' \in \mathcal{X}, L \notin \mathfrak{I}[\![P]\!]\} \quad \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L}\rangle\langle \rho, \tilde{L} : A \to L'\rangle t^{OP}[\sigma', \mathfrak{I}[\![P]\!]] \mid lab[\sigma'] = L', env[\sigma'] \in \mathbf{A}[\![A]\!]\rho, \\ L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \sigma' \in \mathcal{X}, L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

then, assuming $\sigma = t^{OP}[\sigma', \mathfrak{I}[\![P]\!]]$, we obtain:

$$\{\langle \rho, L : A \to L'\rangle\sigma \mid lab[\sigma] = L', env[\sigma] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \\ \rho \in \mathfrak{E}[\![P]\!], \sigma \in t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]], L \notin \mathfrak{I}[\![P]\!]\} \quad \cup$$

$$\{\langle \rho, L : P^T \to \tilde{L}\rangle\langle \rho, \tilde{L} : A \to L'\rangle\sigma \mid lab[\sigma] = L', env[\sigma] \in \mathbf{A}[\![A]\!]\rho, \\ L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \sigma \in t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

where given a trace σ: $env[\sigma] = env[\sigma_0]$ and $env[\langle \rho, C\rangle] = \rho$, while $lab[\sigma] = lab[\sigma_0]$ and $lab[\langle \rho, C\rangle] = lab[\![C]\!]$. By defining $F^{OP}[\![P]\!](t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]])$ as given by the union of the elements obtained by the above computation, we have:

$$F^{OP}[\![P]\!](t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) \stackrel{\text{def}}{=}$$

$$\{\langle \rho, L : A \to L' \rangle \mid L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathcal{L}[\![P]\!], L \notin \mathfrak{I}[\![P]\!]\} \cup$$
$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \mid L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], L' \in \mathcal{L}[\![P]\!], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\} \cup$$
$$\{\langle \rho, L : A \to L' \rangle \sigma \mid lab[\sigma] = L', env[\sigma] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \sigma \in t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]], L \notin \mathfrak{I}[\![P]\!]\} \cup$$
$$\{\langle \rho, L : P^T \to \tilde{L} \rangle \langle \rho, \tilde{L} : A \to L' \rangle \sigma \mid lab[\sigma] = L', env[\sigma] \in \mathbf{A}[\![A]\!]\rho, L : A \to L' \in P, \rho \in \mathfrak{E}[\![P]\!], \sigma \in t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\}$$

Thus, $t^{OP} \circ F^+ = F^{OP} \circ t^{OP}$, and applying the fixpoint transfer theorem we have that $t^{OP}[lfpF^+[\![P]\!], \mathfrak{I}[\![P]\!]]$ can be expressed as $lfpF^{OP}[\![P]\!]$.

Step 3: Let us compute the abstraction $\mathbb{p}^+$ of $F^{OP}[\![P]\!]$ in order to verify the commutation property necessary for fixpoint transfer:

$$\mathbb{p}^+(F^{OP}[t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]]) =$$
$$\{\{L : A \to L'\} \mid L : A \to L' \in P, L' \in \mathcal{L}[\![P]\!] L \notin \mathfrak{I}[\![P]\!]\} \cup$$
$$\{\{L : P^T \to \tilde{L}; \tilde{L} : A \to L'\} \mid L : A \to L' \in P, L' \in \mathcal{L}[\![P]\!], L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New\} \cup$$
$$\{\{L : A \to L'\} \cup \mathbb{p}^+(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) \mid L : A \to L' \in P, L \notin \mathfrak{I}[\![P]\!], \exists C \in \mathbb{p}^+(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) : lab[\![C]\!] = L'\} \cup$$
$$\{\{L : P^T \to \tilde{L}; \tilde{L} : A \to L'\} \cup \mathbb{p}^+(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) \mid L : A \to L' \in P, L \in \mathfrak{I}[\![P]\!], \tilde{L} \in New, \exists C \in \mathbb{p}^+(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]) : lab[\![C]\!] = L'\}$$

Step 4: Defining $\mathbb{F}^{OP}[\![P]\!](\mathbb{p}^+(t^{OP}[\mathcal{X}, \mathfrak{I}[\![P]\!]]))$ as given by the union above, we have that $\mathbb{p}^+ \circ F^{OP}[\![P]\!] = \mathbb{F}^{OP}[\![P]\!] \circ \mathbb{p}^+$, and therefore $\mathbb{p}^+(lfpF^{OP}[\![P]\!]) = lfp\mathbb{F}^{OP}[\![P]\!]$. From the definition of $\mathbb{F}^{OP}$ it is possible to derive an extended iterative algorithm that inserts opaque predicates.

Let us denote with $B \in \wp(\mathbb{C})$ a set of commands composing a possible false path of a true opaque predicate (never executed at run time), and with $lab[\![B]\!]$ the label of the starting point of the execution of B. Let B range over a given collection of programs $\mathfrak{B} \subseteq \wp(\mathbb{C})$, and let $New \subseteq \mathbb{L}$ be a set of "new" program labels. The algorithm **Opaque** considers each command $L : A \to L'$ of the original program, if L is a candidate label for opaque predicate insertion, i.e., if $L \in \mathfrak{I}[\![P]\!]$, the commands $L : P^T \to \tilde{L}$, $\tilde{L} : A \to L'$ and $L : \neg P^T \to lab[\![B]\!]$,

Opaque$(P, \mathfrak{I}[\![P]\!], New, OP, \mathfrak{B})$
$Q = \varnothing$
$T = \{\ C \in P \mid suc[\![C]\!] \in \mathcal{L}[\![P]\!]\ \}$
while there exists an unmarked command $L : A \to L'$ in T do
  mark $L : A \to L'$
  if $L \in \mathfrak{I}[\![P]\!]$
    then take $\tilde{L} \in New$
      $New = New \setminus \tilde{L}$
      let $P^T \in OP$
(∗)   let $B \in \mathfrak{B}$
      $Q = Q \cup \{L : P^T \to \tilde{L}; \tilde{L} : A \to L'\}$
(∗) $Q = Q \cup \{L : \neg P^T \to lab[\![B]\!]\}$
    else $Q = Q \cup \{L : A \to L'\}$
  $T = T \cup \{\ C \in P \mid \exists C' \in T : suc[\![C]\!] = lab[\![C']\!]\ \}$

Fig. 5.2. Opaque predicate insertion algorithm

encoding opaque predicate insertion, are added to set Q (initially empty), otherwise the original command $L : A \to L'$ is added to Q. In particular, command $L : \neg P^T \to lab[\![B]\!]$ encodes the false branch of the true opaque predicate and inserts a fake branch connecting the original program control flow to the flow of the never executed code starting at label $lab[\![B]\!]$. In the end, the set Q corresponds to the obfuscated program. It is clear that $|New| \geq |\mathfrak{I}[\![P]\!]|$. Observe that the lines marked with $(*)$, encoding the insertion of commands forming the false path of the true opaque predicate, have been added manually to $\mathbb{p}^+ \circ t^{OP} \circ S^+$. This happens because the false path of a true opaque predicate is never executed and therefore its commands are not present in the transformed program semantics. In fact, the insertion of an opaque predicate inserts "dead code" in the program (i.e., code that is never executed) and, by definition, the abstraction $\mathbb{p}^+$ cannot return such dead code.

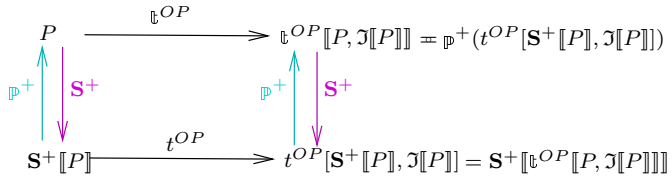


Fig. 5.3. Semantic and Syntactic opaque predicate insertion

Let us denote with $\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$ the extended syntactic transformation corresponding to algorithm **Opaque** reported in Fig. 5.2, and let us report in Fig. 5.3 a schema representing the present situation. Observe that if, on the one hand, $\mathbb{p}^+(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]) \eqq \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$ since they have the same trace seman-

tics, on the other hand $\mathbb{p}^+(t^{OP}[S^+[\![P]\!], \mathfrak{I}[\![P]\!]]) \subset \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$, since the term on the right contains also the commands of the false paths of the inserted true opaque predicates.

5.1.3 Obfuscating behaviour of opaque predicate insertion

In order to study the obfuscating behaviour of opaque predicate insertion we need to define the most concrete property preserved by such transformations. Following Theorem 4.8 we have that the most concrete property preserved by opaque predicate insertion can be characterized as follows:

$$\delta_{t^{OP}} = \bigsqcup_{P \in \mathbb{P}} \left\{ \mathcal{X} \in \wp(\Sigma^+) \,\middle|\, \mathit{Pres}_{P,t^{OP}}(\mathcal{X}) \right\}$$

Where, given $\mathcal{X} \in \wp(\Sigma^+)$, predicate $\mathit{Pres}_{P,t^{OP}}(\mathcal{X}) = \mathit{true}$ if and only if:

$$\forall \mathcal{Y} \subseteq S^+[\![P]\!] : \mathcal{Y} \subseteq \mathcal{X} \Rightarrow \bigcup \left\{ \mathcal{K} \in \wp(\Sigma^+) \,\middle|\, \mathcal{K} = t^{OP}[\mathcal{Y}, \mathfrak{I}[\![P]\!]] \right\} \subseteq \mathcal{X}$$

Meaning that a set of traces $\mathcal{X}$ is "preserved" by opaque predicate insertion if it contains all the traces that can be obtained from traces in $\mathcal{X}$ by inserting opaque predicates from OP at program points indicated by $\mathfrak{I}[\![P]\!]$. As expected, the attacker that observes the concrete semantics of program behaviour is obfuscated by opaque predicate insertion, since $\mathbf{S}^+[\![P]\!] \neq \mathbf{S}^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$, while the attacker observing the denotational semantics of programs is insensitive to opaque predicate insertion, since $\delta_{t^{OP}} \sqsubseteq \mathit{DenSem}$ and $\mathit{DenSem}[\![P]\!] = \mathit{DenSem}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}]\!]]\!]$.

In general, $\mathbf{S}^+[\![P]\!] \neq \mathbf{S}^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$, namely $\mathbf{S}^+[\![P]\!] \neq t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]]$. In particular, the transformed semantics contains all the traces of the original semantics with some extra states denoting opaque predicate execution as described in Fig. 5.1. It is clear that there is no significant information hidden by this obfuscation to attackers knowing the concrete program semantics. In fact, by the observation of the concrete semantics, an attacker can easily derive the set of inserted opaque predicates and deobfuscate the program.

Observe that, knowing the set OP of inserted opaque predicates, we can define the trace transformation $d_{OP} : \wp(\Sigma^+) \to \wp(\Sigma^+)$ that recovers the original program semantics from opaque predicate insertion.

$$d_{OP}(\mathcal{X}) \stackrel{\text{def}}{=} \left\{ d_{OP}(\sigma) \,\middle|\, \sigma \in \mathcal{X} \right\} \quad d_{OP}(\sigma) \stackrel{\text{def}}{=} \epsilon \; d_{OP}(\sigma)$$

$$d_{OP}(\langle \rho, C \rangle \langle \rho', C' \rangle \eta) \stackrel{\text{def}}{=}$$

$$\begin{cases} \langle \rho, C \rangle \; d_{OP}(\langle \rho', C' \rangle \eta) & \text{if } \mathit{act}[\![C]\!] \notin OP \\ d_{OP}(\langle \rho, \mathit{lab}[\![C]\!] : \mathit{act}[\![C']\!] \to \mathit{suc}[\![C']\!] \rangle \eta) & \text{if } \mathit{act}[\![C]\!] \in OP \end{cases}$$

It is not surprising that transformation d_{OP}, given the set of inserted opaque predicates, is able to restore the original program semantics.

Theorem 5.2. $\mathbf{S}^+[\![P]\!] = d_{OP}(\mathbf{S}^+[\![P]\!]) = d_{OP}(t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]])$.

PROOF: Let us assume, as usual, that program P has not been previously obfuscated by opaque predicate insertion. Following the definition of d_{OP} we have that $d_{OP}(\mathbf{S}^+[\![P]\!]) = \mathbf{S}^+[\![P]\!]$, since for each trace $\sigma \in \mathbf{S}^+[\![P]\!] : \forall i : act[\![C_i]\!] \notin OP$. On the other hand $d_{OP}(t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]]) = \{d_{OP}(\eta) \mid \eta \in t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]]\}$. Thus, given $\eta \in t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]]$, there exists $\sigma \in \mathbf{S}^+[\![P]\!]$ such that $\eta = t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]$. In order to conclude the proof we show that $d_{OP}(\eta) = \sigma$, namely that $d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) = \sigma$. In general $\sigma = \mu^1\sigma_i\mu^2\sigma_j\mu^3...\mu^l$, where $\sigma_i = \langle \rho_i, C_i \rangle$ are such that $lab[\![C_i]\!] \in \mathfrak{I}[\![P]\!]$, while μ^i are the portions (even empty) of trace of σ that are unchanged by opaque predicate insertion, that is $\forall \langle \rho, C \rangle \in \mu^i : lab[\![C]\!] \notin \mathfrak{I}[\![P]\!]$. By hypothesis η is obtained from σ by opaque predicate insertion, therefore η has the following structure: $\eta = \mu^1\eta_i^a\eta_i^b\mu^2\eta_j^a\eta_i^b\mu^3...\mu^l$, where $|\eta| = |\sigma| + |\mathfrak{I}[\![P]\!] \cap \{lab[\![C]\!] \mid \langle \rho, C \rangle \in \sigma\}|$ and $\eta_i^a\eta_i^b = \langle \rho_i, L_i : P^T \rightarrow \tilde{L}_i \rangle \langle \rho_i, \tilde{L}_i : A_i \rightarrow L_{i+1} \rangle$. Hence, following the definition of d_{OP} we have:

$$\begin{aligned} d_{OP}(\eta) &= d_{OP}(\mu^1\eta_i^a\eta_i^b\mu^2\eta_j^a\eta_i^i\mu^3...\mu^l) \\ &= \mu^1 d_{OP}(\eta_i^a\eta_i^b\mu^2\eta_j^a\eta_i^b\mu^3...\mu^l) \\ &= \mu^1\sigma_i d_{OP}(\mu^2\eta_j^a\eta_i^b\mu^3...\mu^l) \\ &= \mu^1\sigma_i\mu^2 d_{OP}(\eta_j^a\eta_i^b\mu^3...\mu^l) \\ &= ... \\ &= \mu^1\sigma_i\mu^2\sigma_i\mu^3...\mu^l = \sigma \end{aligned}$$

We have that $d_{OP}(t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]]) = d_{OP}(\{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in \mathbf{S}^+[\![P]\!]\}) = \{d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) \mid \sigma \in \mathbf{S}^+[\![P]\!]\} = \{\sigma \mid \sigma \in \mathbf{S}^+[\![P]\!]\} = \mathbf{S}^+[\![P]\!]$, which concludes the proof.

□

Observe that, by computing transformation d_{OP} on the obfuscated program semantics $\mathbf{S}^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]$, and then deriving the corresponding program through $\mathbb{p}^+$, we obtain exactly the original program P, as shown in Fig. 5.4. This means that, knowing the set OP an attacker can eliminate the inserted opaque predicates, namely $\mathbb{p}^+ \circ d_{OP} \circ \mathbf{S}^+$ acts as a deobfuscation technique.

Example 5.3. Let us consider the trace semantics $\mathbf{S}^+[\![P]\!]$ of program P and a trace $\sigma \in \mathbf{S}^+[\![P]\!]$. Let $\sigma = \langle \rho_0, C_0 \rangle \langle \rho_1, C_1 \rangle \langle \rho_2, C_2 \rangle \langle \rho_3, C_3 \rangle \langle \rho_4, C_4 \rangle$, with commands $C_i = L_i : A_i \rightarrow L_{i+1}$. Let $\mathfrak{I}[\![P]\!] = \{L_1, L_3\}$ be the candidate labels for opaque predicate insertion. The transformed trace is given by:

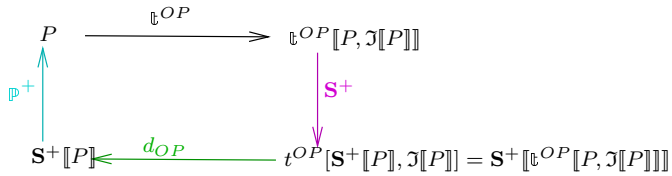


Fig. 5.4. $\mathbb{p}^+ \circ d_{OP} \circ \mathbf{S}^+$ is a deobfuscation technique

$$t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] = \langle \rho_0, C_0\rangle\langle \rho_1, L_1 : P^T \to \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : A_1 \to L_2\rangle\langle \rho_2, C_2\rangle \\ \langle \rho_3, L_3 : P^T \to \tilde{L}_3\rangle\langle \rho_3, \tilde{L}_3 : A_3 \to L_4\rangle\langle \rho_4, C_4\rangle$$

It is easy to show that $d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) = \sigma$, in fact:

$$\begin{aligned} d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) &= d_{OP}(\langle \rho_0, C_0\rangle\langle \rho_1, L_1 : P^T \to \tilde{L}_1\rangle\langle \rho_1, \tilde{L}_1 : A_1 \to L_2\rangle\langle \rho_2, C_2\rangle \\ &\qquad \langle \rho_3, L_3 : P^T \to \tilde{L}_3\rangle\langle \rho_3, \tilde{L}_3 : A_3 \to L_4\rangle\langle \rho_4, C_4\rangle) \\ &= \langle \rho_0, C_0\rangle\langle \rho_1, C_1\rangle\langle \rho_2, C_2\rangle\langle \rho_3, C_3\rangle\langle \rho_4, C_4\rangle \\ &= \sigma \end{aligned}$$

□

Transformation d_{OP} is clearly additive and can therefore be viewed as an abstraction function. It is interesting to observe that, considering the concretization γ_{OP} induced by such abstraction, the property $\gamma_{OP} \circ d_{OP}$ corresponds to the most concrete property preserved by t^{OP}. In fact, knowing OP, the closure $\gamma_{OP} \circ d_{OP}$ observes traces up to opaque predicate insertion, which corresponds to the observation done by $\delta_{t^{OP}}$. In particular, given an obfuscated set of traces $\mathcal{X}$, the deobfuscation $d_{OP}(\mathcal{X}) = \mathcal{Y}$ eliminates the opaque predicates from traces in $\mathcal{X}$, and the concretization $\gamma_{OP}(\mathcal{Y})$ returns the set of all traces that can be obtained from traces in $\mathcal{Y}$ by opaque predicate insertion. This means that requiring $\mathcal{X}$ to be a fixpoint of $\gamma_{OP} \circ d_{OP}$, i.e., $\gamma_{OP}(d_{OP}(\mathcal{X})) = \mathcal{X}$, is equivalent to require that $\mathcal{X}$ satisfies $Pres_{P,t^{OP}}(\mathcal{X})$.

Theorem 5.4. $\gamma_{OP} \circ d_{OP} \in uco(\wp(\Sigma^+))$ and $\gamma_{OP} \circ d_{OP} = \delta_{t^{OP}}$.

PROOF: Function d_{OP} is clearly additive, and $\gamma_{OP} \circ d_{OP} \in uco(\wp(\Sigma^+))$. From Theorem 5.2 it follows that $\gamma_{OP} \circ d_{OP}$ is preserved by t^{OP}, namely $\gamma_{OP}(d_{OP}(\mathbf{S}^+[\![P]\!])) = \gamma_{OP}(d_{OP}(t^{OP}[\mathbf{S}^+[\![P]\!], \mathfrak{I}[\![P]\!]]))$, let us show that it coincides with $\delta_{t^{OP}}$. To do this we have to prove that, given $\mathcal{X} \in \wp(\Sigma^+)$: $\mathcal{X} = \gamma_{OP} \circ d_{OP}(\mathcal{X})$ iff for every program $P \in \mathbb{P}$: $Pres_{P,t^{OP}}(\mathcal{X}) = true$.
($\Rightarrow$) By definition $\gamma_{OP}(d_{OP}(\mathcal{X})) = \{\sigma \mid d_{OP}(\sigma) \subseteq d_{OP}(\mathcal{X})\} = \{\sigma \mid \exists \delta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\delta)\}$. We have to prove that $\forall \mathcal{Y} \subseteq \mathbf{S}^+[\![P]\!] : \mathcal{Y} \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$ the following inclusion holds $\bigcup\{\mathcal{K} \in \wp(\Sigma^+) \mid \mathcal{K} = t^{OP}[\mathcal{Y}, \mathfrak{I}[\![P]\!]]\} \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$. Let $\mathcal{Y} \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$, then $\mathcal{Y} \subseteq \{\sigma \mid \exists \delta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\delta)\}$. This

means that $\forall\sigma \in \mathcal{Y} : \exists\delta \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\delta)$. Following the definition of t^{OP} we have $t^{OP}[\mathcal{Y}, \mathfrak{I}[\![P]\!]] = \{t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \mid \sigma \in \mathcal{Y}\}$. Observe that $\forall\sigma \in \mathcal{Y} : t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\sigma))$, since we have shown that $d_{OP}(t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]) = \sigma$. This means that $\forall\sigma \in \mathcal{Y} : t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\sigma)) = \gamma_{OP}(d_{OP}(\delta)) \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$. Therefore $\forall\sigma \in \mathcal{Y} : t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \in \gamma_{OP}(d_{OP}(\mathcal{X}))$, meaning that $t^{OP}[\mathcal{Y}, \mathfrak{I}[\![P]\!]] \subseteq \gamma_{OP}(d_{OP}(\mathcal{X}))$. The above proof does not depend on the particular program $P \in \mathbb{P}$ considered, meaning that it holds for every program. This means that for any program $P \in \mathbb{P}$ we have that $Pres_{P,t^{OP}}(\mathcal{X}) = true$.
($\Leftarrow$) Assume that for all $P \in \mathbb{P}$: $Pres_{P,t^{OP}}(\mathcal{X}) = true$:

$$\Rightarrow \quad \forall P \in \mathbb{P} : \forall\mathcal{Y} \subseteq \mathbf{S}^{+}[\![P]\!] : \mathcal{Y} \subseteq \mathcal{X} \Rightarrow \bigcup \{\ \mathcal{K} \in \wp(\Sigma^{+}) \,|\, \mathcal{K} = t^{OP}[\mathcal{Y}, \mathfrak{I}[\![P]\!]]\ \} \subseteq \mathcal{X}$$

$$\Rightarrow \quad \forall P \in \mathbb{P} : \forall\mathcal{Y} \subseteq \mathbf{S}^{+}[\![P]\!] : \mathcal{Y} \subseteq \mathcal{X} \Rightarrow \bigcup \{\ \mathcal{K} \in \wp(\Sigma^{+}) \,|\, \mathcal{K} = \{\ t^{OP}[\sigma, \mathfrak{I}[\![P]\!]] \,|\, \sigma \in \mathcal{Y}\ \}\ \} \subseteq \mathcal{X}$$

$$\Rightarrow \quad \forall P \in \mathbb{P} : \forall\sigma \in \mathcal{X} : \bigcup \{\ \eta \,|\, \eta = t^{OP}[\sigma, \mathfrak{I}[\![P]\!]]\ \} \subseteq \mathcal{X}$$

$$\Rightarrow \quad \mathcal{X} = \{\ \delta \,|\, \exists\sigma \in \mathcal{X} : d_{OP}(\sigma) = d_{OP}(\delta)\ \}$$

$$\Rightarrow \quad \mathcal{X} = \gamma_{OP}(d_{OP}(\mathcal{X}))$$

This means that $\delta_{t^{OP}} = \bigsqcup_{P\in\mathbb{P}}\{\mathcal{X} \mid Pres_{P,t^{OP}}(\mathcal{X})\} = \{\gamma_{OP}(d_{OP}(\mathcal{X})) \mid \mathcal{X} \in \wp(\Sigma^{+})\} = \gamma_{OP}(d_{OP}(\wp(\Sigma^{+})))$.

□

5.1.4 Detecting Opaque Predicates

It is clear that the efficiency of transformation d_{OP} in eliminating opaque predicates is based on the knowledge of the set OP. In fact, in the case of opaque predicate insertion, the problem of deobfuscating a program reduces to the ability of detecting opaque predicates. A predicate is opaque if it behaves in the same way in every execution context. Thus, understanding the presence of opaque predicates in a program, means identifying those predicates that evaluate in the same way during every program execution. Given an obfuscated program $\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]$ the set OP of inserted opaque predicates can be characterized by the following definition:

$$OP \stackrel{\text{def}}{=} \left\{ B \,\middle|\, \begin{array}{l} \exists C \in \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!] : act[\![C]\!] = B \\ \forall\sigma \in \mathbf{S}^{+}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!] \\ \forall\langle\rho, C\rangle \in \sigma : (act[\![C]\!] = B) \Rightarrow (\mathbf{B}[\![B]\!]\rho = true) \end{array} \right\} \tag{5.1}$$

This means that having access to the concrete semantics $\mathbf{S}^+[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$ of the obfuscated program, which implies a precise evaluation $\mathbf{B}[\![B]\!]\rho$ of any test action B at any program point, ensures that the resulting set OP contains all the true inserted opaque predicates. Hence, if an attacker observes the concrete execution of an obfuscated program, it can deduce all the necessary information in order to remove opaqueness. In fact, opaque predicate insertion is an obfuscating transformation designed to confuse the control flow of a program. Since program control flow is an abstraction of program trace semantics, it is not surprising that obfuscating the control flow may not cause confusion at the trace semantic level. This is the reason why, in order to better understand the obfuscating behaviour of opaque predicate insertion, we have to consider abstractions of program trace semantics.

In Section 4.3 we have argued how attackers can be modeled as abstract interpretations of the concrete domain of computation of the trace semantics of programs. Thus, it is interesting to investigate the obfuscating behaviour of opaque predicate insertion when attackers have access only to the abstract semantics computed on their abstract domains. Let $\mathbf{S}^{\varphi}$ denote the abstract semantics computed by attacker φ. In particular, if the concrete semantic is given by $\mathbf{S}^+[\![P]\!] = lfpF^+[\![P]\!]$, then the abstract semantics is defined as $\mathbf{S}^{\varphi}[\![P]\!] \stackrel{\text{def}}{=} lfpF^{\varphi}[\![P]\!]$, where F^{φ} is the best correct approximation of the concrete function F^+ on the abstract domain φ. We denote with $\widehat{\mathfrak{E}}$ the set of abstract environments $\hat{\rho} : \mathbb{X} \to \varphi(\mathcal{D}_{\perp})$ that associate abstract values to program variables, with $\hat{\sigma}_i = \langle \hat{\rho}_i, C \rangle$ an abstract state, and with $\hat{\sigma}$ an abstract trace. Moreover, let $\varphi(\wp(\Sigma^+)) = \wp(\widehat{\Sigma}^+)$ be the powerset of abstract traces. It is clear that, in this setting, the most powerful attacker is the one that has access to the most precise description of program behaviour, namely the one that is precise enough to compute the (concrete) program trace semantics $\mathbf{S}^+[\![P]\!]$.

In general, the set OP^{φ} of opaque predicates that an attacker modeled by an abstraction φ is able to identify can be characterized as follows:

$$OP^{\varphi} \stackrel{\text{def}}{=} \left\{ B \;\middle|\; \begin{array}{l} \exists C \in \mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!] : act[\![C]\!] = B \\ \forall \hat{\sigma} \in \mathbf{S}^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!] \\ \forall \langle \hat{\rho}, C \rangle \in \hat{\sigma} : (act[\![C]\!] = B) \Rightarrow (\mathbf{B}^{\varphi}[\![B]\!]\hat{\rho} = true) \end{array} \right\} \tag{5.2}$$

Where $\mathbf{B}^{\varphi}$ denotes the abstract evaluation of boolean expressions. It is clear that, in general, the set of predicates classified as opaque observing the abstract semantics $\mathbf{S}^{\varphi}$ is different from the set of predicates classified as opaque observing program trace semantics $\mathbf{S}^+$, namely $OP^{\varphi} \neq OP$. There are two causes of imprecision, both due to the loss of information implicit in the abstraction process:

– On the one hand, it may happen that φ is not powerful enough to recognize the constantly true value of some opaque predicates, namely there may

exist an opaque predicate P^T such that $P^T \in OP$ while $P^T \notin OP^\varphi$ (see Section 5.3.1 for an example).
– On the other hand, an attacker may classify a predicate as opaque while it is not, namely there may exist a predicate Pr such that $Pr \in OP^\varphi$ while $Pr \notin OP$ (see Section 5.3.3 for an example).

The deobfuscation process that an attacker φ can perform is expressed by the function $d_{OP^\varphi} : \wp(\hat{\Sigma}^+) \to \wp(\hat{\Sigma}^+)$, operating on abstract traces and on set OP^φ of opaque predicates.

$$d_{OP^\varphi}(\hat{\mathcal{X}}) \stackrel{\text{def}}{=} \left\{ \, d_{OP^\varphi}(\hat{\sigma}) \,\middle|\, \hat{\sigma} \in \hat{\mathcal{X}} \, \right\} \quad d_{OP^\varphi}(\hat{\sigma}) \stackrel{\text{def}}{=} \epsilon \; d_{OP^\varphi}(\hat{\sigma})$$

$$d_{OP^\varphi}(\langle \hat{\rho}, C \rangle \langle \hat{\rho}', C' \rangle \hat{\eta}) \stackrel{\text{def}}{=}$$

$$\begin{cases} \langle \hat{\rho}, C \rangle \; d_{OP^\varphi}(\langle \hat{\rho}', C' \rangle \hat{\eta}) & \text{if } act[\![C]\!] \notin OP^\varphi \\ d_{OP^\varphi}(\langle \hat{\rho}, lab[\![C]\!] : act[\![C']\!] \to suc[\![C']\!] \rangle \hat{\eta}) & \text{if } act[\![C]\!] \in OP^\varphi \end{cases}$$

In general, $OP \neq OP^\varphi$ and $\mathbf{S}^\varphi[\![P]\!] \neq d_{OP}(\mathbf{S}^\varphi[\![P]\!]) \neq d_{OP^\varphi}(\mathbf{S}^\varphi[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$, meaning that attacker φ is not able to reverse obfuscation $\mathbb{t}^{OP}$. When attacker φ is not able to disclose the inserted opaque predicates, namely when $\mathbf{S}^\varphi[\![P]\!] \neq \mathbf{S}^\varphi[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$, we say that attacker φ is defeated by the obfuscation (otherwise stated, that the obfuscation is potent with respect to attacker φ). This leads to the following definition of transformation potency.

Definition 5.5. A transformation $\mathbb{t} : \mathbb{P} \to \mathbb{P}$ is *potent* with respect to attacker $\varphi \in uco(\wp(\Sigma^+))$ if there exists $P \in \mathbb{P}$ such that $\mathbf{S}^\varphi[\![P]\!] \neq \mathbf{S}^\varphi[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]$.

It is clear that the above definition of transformation potency is based on the abstract semantics computed by the attacker and not on the abstraction of the concrete semantics as given in Definition 4.3 (where a transformation $\mathbb{t}$ is potent if there exists an abstraction $\varphi \in uco(\wp(\Sigma^+))$ such that $\varphi(\mathbf{S}^+[\![P]\!]) \neq \varphi(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!])$). The two proposed definitions of transformation potency are deeply different and orthogonal. In fact, the results obtained in Chapter 4 referring to Definition 4.3, cannot be projected using Definition 5.5 of potency. However, the two definitions are both useful in understanding the obfuscating behaviour of program transformations. On the one hand, Definition 4.3 can be successfully applied to those obfuscation that have sensitive effects on the concrete program semantics, namely those transformations that cannot be recovered by simply observing the concrete semantics of the obfuscated program (e.g., array merging, variable renaming, substitution of equivalent sequences of instructions, etc.). On the other hand, Definition 5.5 captures the obfuscating behaviour of program transformations that cause minor effects on program trace semantics and that can be recovered by observing the concrete program

semantics (e.g., opaque predicate insertion, code transportation, semantic NOP insertion, etc.).

Fig. 5.5 shows how opaque predicate insertion leaves trace semantics $\mathbf{S}^+$ almost unchanged, while it may significantly modify abstract semantics $\mathbf{S}^\varphi$. In fact, the scheme on the left shows how, considering program trace semantics, it is possible to recover the semantics of the original program. On the other hand, the scheme on the right shows how opaque predicate insertion may prevent attackers from recovering the abstract semantics of the original program.

Fig. 5.5. Trace semantics $\mathbf{S}^+$ and abstract semantics $\mathbf{S}^\varphi$

We are interested in the study of opaque predicate insertion and of the ability of attackers to recover the original program. In particular, it would be interesting to provide a formal characterization of the family of attackers that are able to disclose a given set of opaque predicates. Thus, given a set OP of opaque predicates, we want to characterize the class of attackers φ such that $d_{OP^\varphi}(\mathbf{S}^\varphi[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!]) = d_{OP^\varphi}(\mathbf{S}^\varphi[\![P]\!]) = \mathbf{S}^\varphi[\![P]\!]$. Observe that this equality holds only when attacker φ precisely identifies the set of inserted opaque predicates, namely when $OP = OP^\varphi$. When this happens we have that the obfuscation is harmless with respect to attacker φ, namely that the insertion of opaque predicates from OP is not powerful in contrasting attacker φ. The approach to opaque predicate detection, based on the semantic understanding of code obfuscation and on the abstract domain-based model of attackers, is further investigated in Section 5.3.

5.2 Opaque Predicates Detection Techniques

In this section, we analyze two different approaches to opaque predicates detection. The first one is based on purely dynamic information, while the second one is based on hybrid static/dynamic information [104]. Experimental evaluations on a limited set of inputs show that a dynamic attack removes any opaque predicate, but it has the drawback of classifying many predicates as opaque, while they are not. Thus, dynamic attacks do not provide a trustful solution. Randomized algorithm may be used to eliminate opaque predicates,

in this case the probability of precisely detecting an opaque predicate can be increased by augmenting the number of tries [74]. However randomized algorithms do not give an always trustful solution, but an answer that has a high probability of being precise. On the other hand, experimental evaluations on hybrid static/dynamic attacks show that breaking a single opaque predicate is rather time consuming, and may become infeasible. Next, in Section 5.3, we introduce a novel methodology, based on formal program semantics and semantic approximation by abstract interpretation, to detect and then eliminate opaque predicates. Experimental evaluations show the efficiency of this new method of attack.

5.2.1 Dynamic Attack

Dynamic attackers execute programs with several (but of course not all) different inputs and observe the paths followed after each conditional jump. A dynamic attacker classifies a conditional jump as controlled by a false/true opaque predicate if, during these executions, the false/true path is always taken. Therefore, a dynamic attacker detects all the executed opaque predicates, but, due to the limited set of inputs considered, it may classify a predicate as opaque while it is not, called a *false positive*. Let us measure the false positive rate of a dynamic attacker. We execute the SPECint2000 benchmarks (without adding opaque predicates) with the reference inputs, and then we observe the evaluation of conditional jumps at run time. We use DIOTA[1] [106] to identify conditional jumps that always follow the true path, the false path or take both of them.

	% only flth	*% only jump*	*% both ways*
bzip2	42	23	35
crafty	24	19	57
gap	39	21	40
gcc	36	18	46
gzip	36	24	40
mcf	43	32	25
parser	29	14	57
perlbmk	45	23	33
twolf	39	21	40
vortex	59	26	15
vpr	42	21	38
average	***39***	***22***	***39***

Table 5.1. Execution after conditional jumps

[1] DIOTA: a dynamic instrumentation tool which keeps a running program unaltered at its original location and generate instrumented code on the fly somewhere else.

The benchmarks are listed in Table 5.1. For each benchmark, the percentage of regular conditional jumps that look like false/true opaque predicates are annotated in the first/second column, while the percentage of regular conditional jumps that evaluates in both ways is reported in the third column. Benchmarks do not contain opaque predicates, so that the opaque predicates detected by dynamic attack are all false positives. This experimental evaluations show that a dynamic attacker has an average of false positive rate of 39% and 22%, respectively for false and true opaque predicates. Thus, in average, more than 50% of regular opaque predicates are miss-classified as opaque by dynamic detection techniques. A dynamic attacker may improve these results by using some sort of knowledge of program functionality, in order to generate different inputs that are likely to execute different program paths. However, this preliminary analysis of program functionality may be time consuming. Another possibility, is to generate dynamic test data to improve the condition/decision coverage (CDC)[2]. For complex programs, the CDC is at most 58% [112], so 42% of all conditions will be seen as opaque predicates or dead code by the attacker which is of course incorrect. This leads us to conclude that, in general, dynamic attacks are too imprecise.

5.2.2 Brute Force Attack

In this section we consider an hybrid static/dynamic brute force attack acting on assembly basic blocks[3], where the instructions of the opaque predicate are statically identified (static phase) and are then executed on all possible inputs (dynamic phase). Let us consider the numerical opaque predicate $\forall x \in \mathbb{Z} : 2|(x^2+x)$, that verifies that for every integer value x, x^2+x is always an even number. Observe that the implementation of this opaque predicate decomposes the function $x^2 + x$ into elementary functions such as square x^2 and addition $x + y$. Observe that, once an opaque predicate is inserted in a program, it is possible to further protect the code using transformations meant to mask the opaque predicate itself. For example, hiding constant values by use of address composition or using bit-level operations to hide arithmetic manipulations are obfuscating transformations that mask the inserted opaque predicates. The deobfuscation of these additional transformations and opaque predicate detection are problems that can be studied independently. In the following, we assume that potential additional transformations have already been handled. Moreover, we make the assumption that the instructions (that is, elementary functions) corresponding

[2] Condition/decision coverage measures the percentage of conditional jumps that are executed true at least once and false at least once.

[3] A basic block is a sequence of instructions with a single entry point, single exit point, and no internal branches.

to an opaque predicate are always grouped together, i.e., there are no program instructions between them.

The static phase aims at identifying the instructions corresponding to an opaque predicate. Thus, for each conditional jump j the attacker considers the instruction i immediately preceding j. The dynamic phase then checks whether i and j give rise to an opaque predicate by executing instructions i and j on every possible input. If this is the case the predicate is classified as opaque. Otherwise, the analysis proceeds upward by considering the next instruction preceding i, until an opaque predicate is found or the instructions in the basic block terminate. In this latter case, the predicate is not opaque. The computational effort, measured as number of steps, of this attack is $n^2 * (2^w)^r$, where n is the number of instructions encoding the opaque predicate, r is the number of registers and w is the width of the registers used by the opaque predicate. Consider for example the above true opaque predicate compiled for a 32-bit architecture. The predicate is executed with all possible 2^{32} inputs. This compiled code is then executed under the control of GDB, a well known open-source debugger[4], with all 2^{32} inputs. In particular, $2|(x^2 + x)$ can be written in five x86 instructions, so that for this architecture the computational effort to break this opaque predicate will be $5^2 * 2^{64}$. This is because, during the hybrid attack, two variables are needed as input for the addition, so that there are at most 2 registers taken as input during the attack, i.e., r=2, and the width of these registers is 32 bits, i.e., $w = 32$.

It is interesting to measure the time needed by this attack to detect an opaque predicate. As an example, we consider the opaque predicate $\forall x \in \mathbb{Z} : 2|(x + x)$ and measure the time needed to detect it. In assembly, this opaque predicate in a 16-bit environment consists of three instructions. The execution under control of GDB of these three assembly instructions with all 2^{16} inputs takes 8.83 seconds on a 1.6 GHz Pentium M processor with 1 GB of main memory running RedHat Fedora Core 3. In this experimental evaluation, the static phase has been performed by hand, meaning that the starting instruction of the opaque predicate was given. This leads us to conclude that the hybrid static/dynamic approach is precise although it is noticeably time consuming.

5.3 Breaking Opaque Predicates by Abstract Interpretation

In this section we focus our attention on two particular classes of numerical opaque predicates, and we provide a formal characterization of the family of attackers able to disclose such predicates. The considered numerical predicates are applied in some major software protection techniques as code obfuscation [34], software watermarking [116], tamper-proofing [126] and secure mobile

[4] http://www.gnu.org/software/gdb/

agents [107]. Moreover, this class of opaque predicates is used in recent implementations such as PLTO [134] — a binary rewriting system that transforms a binary program preserving the functionality — LOCO [105] — a tool for binary obfuscating and deobfuscating transformations — and SANDMARK [30] — a tool for software watermarking, tamper proofing and code obfuscation of Java programs (see Table 5.2 for some commonly used opaque predicates). Obviously, the above-mentioned tools are not restricted to the insertion of numerical opaque predicates (for example SANDMARK allows the insertion of predicates based on the difficulty of alias analysis). These classes turn out to be particularly interesting since the ability of an attacker to disclose such predicates can be formulated as a completeness problem in the abstract interpretation field, as shown in Section 5.3.1 and Section 5.3.3. In Section 5.3.2 we report some experimental results showing the improvements in performance of opaque predicate detection algorithms, when the detection methodology takes into account the theoretical results obtained in the Section 5.3.1. This gives an idea of the potential benefits that may come from the proposed formal framework for code obfuscation.

$\forall x, y \in \mathbb{Z} : 7y^2 - 1 \neq x$
$\forall x \in \mathbb{Z} : 3 \mid x^3 - x$
$\forall x \in \mathbb{Z} : 2 \mid x \ \vee \ 8 \mid (x^2 - 1)$
$\forall x \in \mathbb{N} : 14 \mid 3 \cdot 7^{4x-2} + 5 \cdot 4^{2x-1} - 5$
$\forall x \in \mathbb{Z} : \sum_{i=1,2 mod(i,2) \neq 0}^{2x-1} i = x$

Table 5.2. Commonly used opaque predicates

5.3.1 Breaking Opaque Predicates $n|f(x)$

Let us consider numerical true opaque predicates of the form:

$$\forall x \in \mathbb{Z} : n|f(x)$$

These predicates are based on a function $f : \mathbb{Z} \to \mathbb{Z}$ that always returns a value that is a multiple of $n \in \mathbb{N}$. This class of opaque predicates is used in major obfuscating tools such as SANDMARK [30] and LOCO [105], and in the software watermarking algorithm by Arboit [6], recently implemented by Myles and Collberg [116].

In order to precisely detect that predicate $n|f(x)$ is opaque one needs to check the *concrete test*, denoted as CT^f and defined as follows:

$$\mathrm{CT}^f \stackrel{\text{def}}{=} \forall x \in \mathbb{Z} : f(x) \in n\mathbb{Z}$$

where $n\mathbb{Z}$ denotes the set of integers that are multiples of $n \in \mathbb{N}$. Observe that, the set of predicates satisfying the concrete test coincides with the set

OP of predicates characterized by (5.1). Our goal is to devise an abstract interpretation-based method which allows us to perform the test of opaqueness for f on a suitable abstract domain. As observed in Section 2.2, abstraction can be equivalently encoded as a closure operator $\varphi \in uco(\wp(\mathbb{Z}))$, or as an abstract domain $A \cong \varphi(\wp(\mathbb{Z}))$. In this section we prefer the abstract domain representation $A \in uco(\wp(\mathbb{Z}))$, and we denote with $\alpha_A : \wp(\mathbb{Z}) \to A$ and $\gamma_A : A \to \wp(\mathbb{Z})$ the corresponding abstraction and concretization functions. In particular, we are interested in abstract domains which are able to represent precisely the property of being a multiple of n, i.e., abstract domains $A \in uco(\wp(\mathbb{Z}))$ such that there exists some $a_n \in A$ such that $\gamma_A(a_n) = n\mathbb{Z}$. Let $f^\sharp : A \to A$ be an abstract function that approximates f on A. Then, the *abstract test* on A is defined as follows:

$$\mathrm{AT}_A^{f^\sharp} \stackrel{\text{def}}{=} \forall x \in \mathbb{Z} : f^\sharp(\alpha_A(\{x\})) \leq_A a_n$$

Observe that, the set of predicates satisfying the abstract test on A, corresponds to the set OP^φ (also denoted OP^A) of predicates characterized by (5.2). It is clear that the precision of the abstract test strongly depends on the considered abstract domain. In particular, we have that an abstract test is sound when the satisfaction of the abstract test implies the satisfaction of the concrete one, and complete when the converse holds.

Definition 5.6. Given an opaque predicate $\forall x \in \mathbb{Z} : n | f(x)$ and an abstract domain $A \in uco(\wp(\mathbb{Z}))$, we say that:

- $\mathrm{AT}_A^{f^\sharp}$ is *sound* when $\mathrm{AT}_A^{f^\sharp} \Rightarrow \mathrm{CT}^f$
- and $\mathrm{AT}_A^{f^\sharp}$ is *complete* when $\mathrm{CT}^f \Rightarrow \mathrm{AT}_A^{f^\sharp}$

When the abstract test $\mathrm{AT}_A^{f^\sharp}$ is both sound and complete we say that the attack $\langle A, f^\sharp \rangle$ (or simply A when $f^\sharp$ is clear from the context) *breaks* the opaque predicate $\forall x \in \mathbb{Z} : n | f(x)$. The following result shows that when the abstract function $f^\sharp$ is a sound (resp. $\mathcal{B}$-complete) approximation of f on singletons, then the corresponding abstract test $\mathrm{AT}_A^{f^\sharp}$ is sound (resp. complete).

Theorem 5.7. Consider an attacker $A \in uco(\wp(\mathbb{Z}))$ such that there exists $a_n \in A$ with $\gamma_A(a_n) = n\mathbb{Z}$.

(1) If $f^\sharp$ is sound approximation of f on the singletons, that is $\forall x \in \mathbb{Z}$, $\alpha_A(\{f(x)\}) \leq_A f^\sharp(\alpha_A(\{x\}))$, then $\mathrm{AT}_A^{f^\sharp}$ is sound.
(2) If $f^\sharp$ is $\mathcal{B}$-complete approximation of f on the singletons, that is $\forall x \in \mathbb{Z}$, $\alpha_A(\{f(x)\}) = f^\sharp(\alpha_A(\{x\}))$, then $\mathrm{AT}_A^{f^\sharp}$ is complete.

PROOF: (1) Assume the satisfaction of the abstract test $\mathrm{AT}_A^{f^\sharp}$, namely that $\forall x \in \mathbb{Z} : f^\sharp(\alpha_A(\{x\})) \leq_A a_n$, then for any $x \in \mathbb{Z}$:

$$f(x) \subseteq \gamma_A(\alpha_A(\{f(x)\})) \subseteq \gamma_A(f^\sharp(\alpha_A(\{x\}))) \subseteq \gamma_A(a_n) = n\mathbb{Z}$$

thus $\forall x \in \mathbb{Z} : f(x) \subseteq n\mathbb{Z}$ and the concrete test CT^f holds.
(2) Assume the satisfaction of the concrete test CT^f, i.e., $\forall x \in \mathbb{Z} : f(x) \subseteq n\mathbb{Z}$. Function $f^\sharp$ is $\mathcal{B}$-complete on singletons by hypothesis and therefore for any $x \in \mathbb{Z}$:

$$f^\sharp(\alpha_A(\{x\})) = \alpha_A(\{f(x)\}) \subseteq \alpha_A(n\mathbb{Z}) = a_n$$

□

Thus, the key point in order to detect an opaque predicate $\forall x \in \mathbb{Z} : n \mid f(x)$, is to design a suitable abstract domain A together with a $\mathcal{B}$-complete approximation $f^\sharp$ of f.

Abstract Functions

We already observed in Section 5.2.2 that a function $f : \mathbb{Z} \to \mathbb{Z}$ is decomposed into elementary functions, i.e., assembly instructions within some basic block. Following the same approach, let us assume that the function f can be expressed as a composition of elementary functions, namely $f = \lambda x.h(g_1(x, ..., x), ..., g_k(x, ..., x))$ where $h : \mathbb{Z}^k \to \mathbb{Z}$ and $g_i : \mathbb{Z}^{n_i} \to \mathbb{Z}$. More in general, each g_i can be further decomposed into elementary functions. For example, $f(x) = x^2 + x$ is decomposed as $h(g_1(x), g_2(x))$ where $h(x, y) = x + y$, $g_1(x) = x^2$ and $g_2(x) = x$. Let us consider the pointwise extensions of the elementary functions, which are still denoted, with a slight abuse of notation, by $h : \wp(\mathbb{Z})^k \to \wp(\mathbb{Z})$ and $g_i : \wp(\mathbb{Z})^{n_i} \to \wp(\mathbb{Z})$, and let us denote their composition by

$$F \stackrel{\text{def}}{=} \lambda X.h(g_1(X, ..., X), ..., g_k(X, ..., X)) : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$$

For example, for the above decomposition $f(x) = x^2 + x = h(g_1(x), g_2(x))$, we have that $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is as follows: $F(X) = \{y^2 + z \mid y, z \in X\}$. Observe that F does not coincide with the pointwise extension f^p of f, e.g., $F(\{1, 2\}) = \{2, 3, 5, 6\}$ while $f^p(\{1, 2\}) = \{2, 6\}$. Let us also notice that F on singletons coincides with f, namely for any $x \in \mathbb{Z}$, $F(\{x\}) = f(x)$. Thus, the concrete test CT^f can be equivalently formulated as $\forall x \in \mathbb{Z} : F(\{x\}) \subseteq n\mathbb{Z}$.

Let $A \in uco(\wp(\mathbb{Z}))$ be an abstract domain such that there exists some $a_n \in A$ with $\gamma_A(a_n) = n\mathbb{Z}$. The attacker A approximates the computation of function $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ in a step by step fashion, meaning that A approximates every elementary function composing F. Thus, the abstract function $F^\sharp : A \to A$ is defined as the composition of the best correct approximations h^A and g_i^A on A of the elementary functions, namely:

$$\begin{aligned} F^\sharp(a) &\stackrel{\text{def}}{=} \alpha_A(h(\gamma_A(\alpha_A(g_1(\gamma_A(a), ..., \gamma_A(a)))), ..., \gamma_A(\alpha_A(g_k(\gamma_A(a), ..., \gamma_A(a)))))) \\ &= h^A(g_i^A(a), ..., g_k^A(a)) \end{aligned}$$

When the abstract test $\mathrm{AT}_A^{F^\sharp}$ for $F^\sharp$ on A holds, the attacker modeled by the abstract domain A classifies the predicate $n|f(x)$ as opaque. It turns out that $F^\sharp$ is a correct approximation of F on A, namely $\alpha_A \circ F \sqsubseteq_A F^\sharp \circ \alpha_A$, and this guarantees the soundness of the abstract test $\mathrm{AT}_A^{F^\sharp}$.

Corollary 5.8. $\mathrm{AT}_A^{F^\sharp}$ is sound.

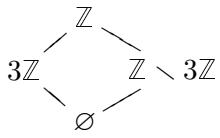
PROOF: We first show that $F^\sharp : A \to A$ is a sound approximation of $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, namely $\forall X \in \wp(\mathbb{Z}) : \alpha_A(F(X)) \leq_A F^\sharp(\alpha_A(X))$. In fact for any $X \in \wp(\mathbb{Z})$:

$$\begin{array}{rcl}
\alpha_A(F(X)) & = & \alpha_A(h(g_1(X,...,X),...,g_k(X,...,X))) \\
& \leq_A & \alpha_A(h(\gamma_A(g_1(X,...,X)),...,\gamma_A(g_k(X,...,X)))) \\
& \leq_A & \alpha_A(h(\gamma_A(\alpha_A(g_1(\gamma_A(\alpha_A(X)),...,\gamma_A(\alpha_A(X)))),..., \\
& & \quad \gamma_A(\alpha_A(g_k(\gamma_A(\alpha_A(X)),...,\gamma_A(\alpha_A(X))))))) \\
& = & F^\sharp(\alpha_A(X))
\end{array}$$

In particular this means that $\forall \{x\} \in \wp(\mathbb{Z}) : \alpha_A(F(\{x\})) \leq_A F^\sharp(\alpha_A(\{x\}))$, i.e., $\forall x \in \mathbb{Z} : \alpha_A(\{f(x)\}) \leq_A F^\sharp(\alpha_A(\{x\}))$. Thus $F^\sharp$ is a sound approximation of f on the singletons and therefore by point (1) of Theorem 5.7 the abstract test $\mathrm{AT}^{F^\sharp}$ is sound.

□

Consider for example the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 - x)$, and the abstract domain A_{3+} in the figure below. A_{3+} precisely represents the property of being a multiple of 3, i.e., $3\mathbb{Z}$, and its negation, i.e., $\mathbb{Z} \smallsetminus 3\mathbb{Z}$.



In this case, $f(x) = x^3 - x = h(g_1(x), g_2(x))$ where $h(x, y) = x - y$, $g_1(x) = x^3$ and $g_2(x) = x$, so that $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is given by $F(X) = \{y^3 - z \mid y, z \in X\}$. Hence, it turns out that $F^\sharp(3\mathbb{Z}) = 3\mathbb{Z}$ while $F^\sharp(\mathbb{Z} \smallsetminus 3\mathbb{Z}) = \mathbb{Z}$. Here, the abstract test $\mathrm{AT}_{A_{3+}}^{F^\sharp}$ is sound but not $\mathcal{B}$-complete, because $F^\sharp : A_{3+} \to A_{3+}$ is a sound but not complete approximation of f on the singletons. In fact, for $\{2\} \in \wp(\mathbb{Z})$, it turns out that $\alpha_{A_{3+}}(\{f(2)\}) = \alpha_{A_{3+}}(\{6\}) = 3\mathbb{Z}$ while $F^\sharp(\alpha_{A_{3+}}(\{2\})) = F^\sharp(\mathbb{Z} \smallsetminus 3\mathbb{Z}) = \mathbb{Z}$. Thus, the abstract test $\mathrm{AT}_{A_{3+}}^{F^\sharp}$, i.e., $\forall x \in \mathbb{Z} : F^\sharp(\alpha_{A_{3+}}(\{x\})) \leq 3\mathbb{Z}$ does not hold even if CT^f does. This means that $OP^A \subseteq OP$, namely that the predicates that satisfy the abstract test are actually opaque, while there may be predicates that are opaque and that are not detected by the abstract test. Thus, in general, $\mathrm{AT}_A^{F^\sharp}$ is sound but not complete, meaning that the attacker $\langle A, F^\sharp \rangle$ is not able to break the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

Recall that abstract domain completeness is preserved by function composition [61], i.e., if an abstract domain A is complete for f and g then A is complete for $f \circ g$ as well. As a consequence, if an abstract domain A is $\mathcal{B}$-complete for the elementary functions h and g_i that decompose F then A is $\mathcal{B}$-complete also for their composition F. It turns out that $\mathcal{B}$-completeness of an abstract domain A with respect to the elementary functions composing F guarantees that the attacker A is able to break the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

Corollary 5.9. Consider an abstract domain $A \in uco(\wp(\mathbb{Z}))$ such that $\exists a_n \in A$ with $\gamma_A(a_n) = n\mathbb{Z}$. If A is $\mathcal{B}$-complete for the elementary functions h and g_i composing F then $\langle A, F^\sharp \rangle$ breaks the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

PROOF: If A is $\mathcal{B}$-complete for h and g_i then it is also $\mathcal{B}$-complete for their composition $F = \lambda X.h(g_1(X, ..., X), ..., g_k(X, ..., X))$. When A is $\mathcal{B}$-complete for $h : \wp(\mathbb{Z})^k \to \wp(\mathbb{Z})$ and $g_i : \wp(\mathbb{Z})^{n_1} \to \wp(\mathbb{Z})$, it means that the best correct approximations of h and g_i respectively are $\mathcal{B}$-complete approximation, namely:

- $\forall X_i \in \wp(\mathbb{Z}) : \alpha_A(h(X_1, ..., X_K)) = \alpha_A(h(\gamma_A(\alpha_A(X_1))), ..., \gamma_A(\alpha_A(X_k)))$
- $\forall Y_i \in \wp(\mathbb{Z}) : \alpha_A(g_i(Y_1, ..., Y_{n_i})) = \alpha_A(g_i(\gamma_A(\alpha_A(Y_1))), ..., \gamma_A(\alpha_A(Y_{n_i})))$

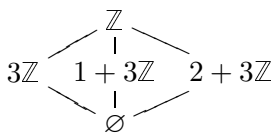
Thus the best correct approximation of F in A is $\mathcal{B}$-complete, i.e., $\forall X \in \wp(\mathbb{Z}) : \alpha_A(F(X)) = F^A(\alpha_A(X))$. It turns out that when the domain is $\mathcal{B}$-complete for h and g_i, then the best correct approximation of F on A coincides with the composition of the best correct approximations of h and g_i, namely $F^\sharp = F^A$. In fact for all $S \in A \in uco(\wp(\mathbb{Z}))$:

$$\begin{aligned} F^\sharp(S) &= \alpha_A(h(\gamma_A(\alpha_A(g_1(\gamma_A(S)), ..., \gamma_A(S))))), ..., \gamma_A(\alpha_A(g_k(\gamma_A(S)), ..., \gamma_A(\alpha_A(X)))))) \\ &= \alpha_A(h(g_1(\gamma_A(S))), ..., \gamma_A(S))), ..., (g_k(\gamma_A(S))), ..., \gamma_A(S))))) \\ &= \alpha_A(F(\gamma_A(S))) = F^A(S) \end{aligned}$$

This means that $F^\sharp$ is a $\mathcal{B}$-complete approximation of F, namely $\forall X \in \wp(\mathbb{Z}) : F^\sharp(\alpha_A(X)) = \alpha_A(F(X))$. In particular $\forall \{x\} \in \wp(\mathbb{Z}) : F^\sharp(\alpha_A(\{x\})) = \alpha_A(F(\{x\})) = \alpha_A(\{f(x)\})$, meaning that $F^\sharp$ is a $\mathcal{B}$-complete approximation of f on the singletons. Therefore by point (2) of Theorem 5.7, the abstract test $\mathrm{AT}_A^{F^\sharp}$ is complete, meaning that the attacker A breaks the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

□

Let us consider the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 - x)$ and the abstract domain 3-*arity* represented in the following figure.



The function $f(x) = x^3 - x$ is decomposed as $h(g_1(x), g_2(x))$ where $h(x, y) = x - y$, $g_1(x) = x^3$ and $g_2(x) = x$. It turns out that the abstract domain 3-*arity* is $\mathcal{B}$-complete for the pointwise extensions of h, g_1 and g_2, i.e., $\lambda\langle X, Y\rangle.X - Y$, $\lambda X.X^3$ and $\lambda X.X$, and therefore, by Corollary 5.9, the attacker 3-*arity* is able to break the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 + x)$.

Lemma 5.10. 3-*arity* is $\mathcal{B}$-complete for $\lambda X.X^3$, $\lambda X.X$ and $\lambda\langle X, Y\rangle.X - Y$.

PROOF: It is easy to verify that given $X \subseteq 3\mathbb{Z}(1+3\mathbb{Z}, 2+3\mathbb{Z})$ then $X^3 \subseteq 3\mathbb{Z}(1+3\mathbb{Z}, 2+3\mathbb{Z})$ respectively. We can see that 3-*arity* is $\mathcal{B}$-complete for $g(X) = X^3$, in fact: if $X \subseteq 3\mathbb{Z}$ then 3-$arity(g(3\text{-}arity(X))) = 3\text{-}arity(g(3\mathbb{Z})) = 3\mathbb{Z}$, and 3-$arity(g(X)) = 3\mathbb{Z}$, and the same holds for $X \subseteq 1 + 3\mathbb{Z}$ and $X \subseteq 2 + 3\mathbb{Z}$. At the end we have to consider also the case in which 3-$arity(X) = \mathbb{Z}$ and the one when $X = \emptyset$. If 3-$arity(X) = \mathbb{Z}$ then 3-$arity(g(3\text{-}arity(X))) = 3\text{-}arity(g(\mathbb{Z})) = \mathbb{Z}$ and 3-$arity(g(X)) = \mathbb{Z}$, while if $X = \emptyset$ then 3-$arity(X) = \bot$ and therefore 3-$arity(g(3\text{-}arity(X))) = 3\text{-}arity(g(\bot)) = \bot$ and 3-$arity(g(\emptyset)) = \bot$.
Now we need to prove that 3-*arity* is complete for the function $h(X, Y) = X - Y$, namely we have to check that for every possible abstractions of X and Y in 3-*arity* then 3-$arity(h(3\text{-}arity(X), 3\text{-}arity(Y))) = 3\text{-}arity(h(X, Y))$. This proof is done by analyzing all the possible cases:

- $X \subseteq 3\mathbb{Z}$, $Y \subseteq 3\mathbb{Z}$, and $X, Y \neq \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity(3\mathbb{Z} - 3\mathbb{Z}) = 3\mathbb{Z}$ and 3-$arity(X - Y) = 3\mathbb{Z}$
- $X \subseteq 1 + 3\mathbb{Z}$, $Y \subseteq 3\mathbb{Z}$, and $X, Y \neq \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity((1 + 3\mathbb{Z}) - 3\mathbb{Z}) = 1 + 3\mathbb{Z}$ and 3-$arity(X - Y) = 1 + 3\mathbb{Z}$
- $X \subseteq 2 + 3\mathbb{Z}$, $Y \subseteq 3\mathbb{Z}$, and $X, Y \neq \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity((2 + 3\mathbb{Z}) - 3\mathbb{Z}) = 2 + 3\mathbb{Z}$ and 3-$arity(X - Y) = 2 + 3\mathbb{Z}$
- $X = \emptyset$ and $Y = \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity(\bot - \bot) = \bot$ and 3-$arity(X - Y) = \bot$
- $X \subseteq 3\mathbb{Z}$, $Y = \emptyset$, and $X \neq \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity(3\mathbb{Z} - \bot) = 3\mathbb{Z}$ and 3-$arity(X - Y) = 3\mathbb{Z}$
- $X \subseteq 1 + 3\mathbb{Z}$, $Y = \emptyset$, and $X \neq \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity((1 + 3\mathbb{Z}) - \bot) = 1 + 3\mathbb{Z}$ and 3-$arity(X - Y) = 1 + 3\mathbb{Z}$
- $X \subseteq 2 + 3\mathbb{Z}$, $Y = \emptyset$, and $X \neq \emptyset$:
  3-$arity(3\text{-}arity(X) - 3\text{-}arity(Y)) = 3\text{-}arity((2 + 3\mathbb{Z}) - \bot) = 2 + 3\mathbb{Z}$ and 3-$arity(X - Y) = 2 + 3\mathbb{Z}$

- $X \subseteq 2+3\mathbb{Z}$, $Y \subseteq 1+3\mathbb{Z}$, and $X, Y \neq \emptyset$:
  3-$arity$(3-$arity$(X) − 3-$arity$(Y)) = 3-$arity$($(2+3\mathbb{Z}) - (1+3\mathbb{Z})$) = $1+3\mathbb{Z}$ and
  3-$arity$($X - Y$) = $1+3\mathbb{Z}$
- $X \subseteq \mathbb{Z}$, $Y \subseteq 3\mathbb{Z}$, $X, Y \neq \emptyset$ and $X \not\subseteq 3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}$:
  3-$arity$(3-$arity$(X)−3-$arity$(Y)) = 3-$arity$($\mathbb{Z}-3\mathbb{Z}$) = $\mathbb{Z}$ and 3-$arity$($X-Y$) = $\mathbb{Z}$
- $X \subseteq \mathbb{Z}$, $Y \subseteq 1+3\mathbb{Z}$, $X, Y \neq \emptyset$ and $X \not\subseteq 3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}$:
  3-$arity$(3-$arity$(X)−3-$arity$(Y)) = 3-$arity$($\mathbb{Z}-(1+3\mathbb{Z})$) = $\mathbb{Z}$ and 3-$arity$($X-Y$) = $\mathbb{Z}$
- $X \subseteq \mathbb{Z}$, $Y \subseteq 2+3\mathbb{Z}$, $X, Y \neq \emptyset$ and $X \not\subseteq 3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}$:
  3-$arity$(3-$arity$(X)−3-$arity$(Y)) = 3-$arity$($\mathbb{Z}-(2+3\mathbb{Z})$) = $\mathbb{Z}$ and 3-$arity$($X-Y$) = $\mathbb{Z}$
- $X \subseteq \mathbb{Z}$, $Y \subseteq \mathbb{Z}$, $X, Y \neq \emptyset$ and $X, Y \not\subseteq 3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}$:
  3-$arity$(3-$arity$(X) − 3-$arity$(Y)) = 3-$arity$($\mathbb{Z}-\mathbb{Z}$) = $\mathbb{Z}$ and 3-$arity$($X-Y$) = $\mathbb{Z}$
- $X \subseteq \mathbb{Z}$, $Y = \emptyset$, $X \neq \emptyset$ and $X \not\subseteq 3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}$:
  3-$arity$(3-$arity$(X)−3-$arity$(Y)) = 3-$arity$($\mathbb{Z}-\bot$) = $\mathbb{Z}$ and 3-$arity$($X-Y$) = $\mathbb{Z}$
- $X \subseteq 1+3\mathbb{Z}$, $Y \subseteq 1+3\mathbb{Z}$, $X, Y \neq \emptyset$
  3-$arity$(3-$arity$(X) − 3-$arity$(Y)) = 3-$arity$($(1+3\mathbb{Z}) - (1+3\mathbb{Z})$) = $3\mathbb{Z}$ and
  3-$arity$($X - Y$) = $3\mathbb{Z}$
- $X \subseteq 2+3\mathbb{Z}$, $Y \subseteq 2+3\mathbb{Z}$, $X, Y \neq \emptyset$
  3-$arity$(3-$arity$(X) − 3-$arity$(Y)) = 3-$arity$($(2+3\mathbb{Z}) - (2+3\mathbb{Z})$) = $3\mathbb{Z}$ and
  3-$arity$($X - Y$) = $3\mathbb{Z}$

Observe that these are all the cases we have to consider since the remaining once follow for semi-commutativity, i.e., $X - Y = -(Y - X)$.

□

Designing Domains for Breaking Opaque Predicates

In the following we show how $\mathcal{B}$-completeness domain refinement can be used to derive models of attackers which are able to break a given opaque predicate. Let us consider the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 - x)$ and the attacker $A_3 \stackrel{\text{def}}{=} \{\mathbb{Z}, 3\mathbb{Z}\}$, that is the minimal abstract domain which represents precisely the property of being a multiple of 3. Recall that the function $f(x) = x^3 - x$ is decomposed as $h(g_1(x), g_2(x))$ where $h(x, y) = x - y$, $g_1(x) = x^3$ and $g_2(x) = x$. It turns out that A_3 is not able to break the above opaque predicate, since $F^\sharp : A_3 \to A_3$ is not a $\mathcal{B}$-complete approximation of f on singletons. In fact, consider $\{2\} \in \wp(\mathbb{Z})$, it turns out that $\alpha_{A_3}(\{f(2)\}) = \alpha_{A_3}(\{6\}) = 3\mathbb{Z}$ while $F^\sharp(\alpha_{A_3}(\{2\})) = F^\sharp(\mathbb{Z}) = \mathbb{Z}$. Corollary 5.9 does not apply here because A_3 is $\mathcal{B}$-complete for g_1 and g_2 but not for h. However, as recalled in Section 2.2,

completeness can be obtained by a domain refinement. Thus, we systematically transform A_3 by the $\mathcal{B}$-completeness domain refinement with respect to $h = \lambda\langle X, Y\rangle.X - Y$. We obtain the abstract domain $\mathcal{R}^{\mathcal{B}}_h(A_3)$ that models an attacker which is able to break $\forall x \in \mathbb{Z} : 3|(x^3 - x)$. As recalled in Section 2.2, the application of the $\mathcal{B}$-completeness domain refinement adds to $A_{3\mathbb{Z}}$ the maximal inverse images under h of all its elements until a fixpoint is reached, that is for any fixed $X \subseteq \mathbb{Z}$ and a belonging to the current abstract domain, we iteratively add the following sets of integers: $\max\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq a\}$. It is not hard to verify that the following elements provide exactly the minimal amount information to add to A_3 in order to make it complete for h.

- if $X = \{0\}$ then: $\max\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq 3\mathbb{Z}\} = 3\mathbb{Z}$
- if $X = \{1\}$ then: $\max\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq 3\mathbb{Z}\} = 1 + 3\mathbb{Z}$
- if $X = \{2\}$ then: $\max\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq 3\mathbb{Z}\} = 2 + 3\mathbb{Z}$

Therefore, $\mathcal{R}^{\mathcal{B}}_h(A_3) = \{\mathbb{Z}, 3\mathbb{Z}, 1 + 3\mathbb{Z}, 2 + 3\mathbb{Z}, \varnothing\} = 3\text{-}arity$. Note that we are able to systematically obtain attacker 3-*arity*, which is able to break the opaque predicate, through a $\mathcal{B}$-completeness refinement of the minimal abstract domain A_3.

It turns out that given $n \in \mathbb{N}$, the abstract domain n-*arity*, in Fig. 5.6, is $\mathcal{B}$-complete for addition, difference and, for $k \in \mathbb{N}$, k-power (i.e., $\lambda X.X^k$). Therefore, by Corollary 5.9, the attacker n-*arity* breaks the opaque predicates $\forall x \in \mathbb{Z}, n|f(x)$, where f is a polynomial function. The abstract domain n-*arity* turns out to be an instance of a more general domain designed by Granger to represent integer congruences [66].

Theorem 5.11. The attacker n-*arity* breaks all the opaque predicates of the following form: $\forall x \in \mathbb{Z} : \ n|f(x)$, where $f(x)$ is a polynomial function.

PROOF: Follows from Corollary 5.9 since n-*arity* is $\mathcal{B}$-complete for addition, difference and k-power (x^k), with $k \in \mathbb{N}$, which are the elementary functions composing f.

- Addition $\forall X, Y \in \wp(\mathbb{Z}) : n\text{-}arity(n\text{-}arity(X) + n\text{-}arity(Y)) = n\text{-}arity(X + Y)$
  Let $i, j \in [0, n-1]$ and let $X, Y \in \wp(\mathbb{Z})$ such that: $n\text{-}arity(X) = i + n\mathbb{Z}$, $n\text{-}arity(Y) = j + n\mathbb{Z}$, then: $n\text{-}arity(i + n\mathbb{Z} + j + n\mathbb{Z}) = n\text{-}arity(i + j + n\mathbb{Z}) = (i + j) \bmod n + n\mathbb{Z}$ and $n\text{-}arity(X + Y) = (i + j) \bmod n + n\mathbb{Z}$.
- Difference: same as for addition
- Power: $\forall X \in \wp(\mathbb{Z}), k \in \mathbb{N} : n\text{-}arity(n\text{-}arity(X)^k) = n\text{-}arity(X^k)$
  Let $i \in [0, n-1]$ and let $X \in \wp(\mathbb{Z})$ such that $n\text{-}arity(X) = i + n\mathbb{Z}$ then: $n\text{-}arity((i+n\mathbb{Z})^k) = n\text{-}arity((i+n\mathbb{Z})(i+n\mathbb{Z})^{k-1}) = i+n\mathbb{Z}$ and $n\text{-}arity(X^k) = n\text{-}arity(xx^{k-1}) = i + n\mathbb{Z}$.

□

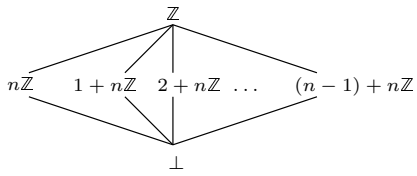


Fig. 5.6. Abstract domain *n-arity*

Breaking Opaque Predicates $P(f(x))$

In the following we generalize the result obtained for opaque predicates of the form $\forall x \in \mathbb{Z} : n|f(x)$ to a wider class of opaque predicates. Let us now consider the class $P(f(x))$ of opaque predicates where each predicate has the following form: $\forall x \in \mathbb{Z} : f(x) \subseteq P$, where $P \subseteq \mathbb{Z}$ is any property on integers numbers and $f : \mathbb{Z} \to \mathbb{Z}$. It is possible to generalize the results of Theorem 5.7, Corollary 5.8 and Corollary 5.9, to opaque predicates in $P(f(x))$. This is simply done by replacing the property $n\mathbb{Z}$ of being a multiple of n, with a general property P over integers. This allows us to provide a formal methodology for designing abstract domains that model attackers able to break opaque predicates in $P(f(x))$. Let $\forall x \in \mathbb{Z} : f(x) \subseteq P$ be an opaque predicate and let us consider the minimal abstract domain A_P that represents precisely the property P, i.e., $A_P \stackrel{\text{def}}{=} \{\mathbb{Z}, P\}$. As above, we assume that the function f can be expressed as a composition of elementary functions, namely $f = \lambda x.h(g_1(x, ..., x), ..., g_k(x, ..., x))$ where $h : \mathbb{Z}^k \to \mathbb{Z}$ and $g_i : \mathbb{Z}^{n_i} \to \mathbb{Z}$. Then, we compute the $\mathcal{B}$-completeness domain refinement of A_P with respect to the set of elementary functions composing f, namely $\mathcal{R}^{\mathcal{B}}_{h,g_1,...,g_k}(A_P)$. It turns out that the refined domain is able to break the opaque predicate $\forall x \in \mathbb{Z} : f(x) \subseteq P$.

Theorem 5.12. The attacker modeled by the abstract domain $\mathcal{R}^{\mathcal{B}}_{h,g_1,...,g_k}(A_P)$ breaks the opaque predicate $\forall x \in \mathbb{Z} : f(x) \subseteq P$.

PROOF: The abstract domain $\mathcal{R}^{\mathcal{B}}_{h,g_1,...,g_k}(A_P)$ is $\mathcal{B}$-complete for the elementary functions h and g_i composing function f. Thus the result follows form Corollary 5.9 where the property $n\mathbb{Z}$ of being a multiple of n is replaced by the general property P over integers.

□

Thus, $\mathcal{B}$-completeness domain refinement provides here a systematic methodology for designing attackers that are able to break opaque predicates of the general form: $\forall x \in \mathbb{Z} : f(x) \subseteq P$.

It is clear that, the previous result is independent from the choice of the concrete domain $\mathbb{Z}$ and can be extended to a general domain of computation *Dom*.

Corollary 5.13. Consider an opaque predicate $\forall x \in Dom$: $f(x) \subseteq P$, with function $f : Dom \rightarrow Dom$, $f = h(g_1(x, ..., x), ..., g_k(x, ..., x))$, and $P \subseteq Dom$. The abstract domain $\mathcal{R}^{\mathcal{B}}_{h,g_1,...,g_k}(\{Dom,P\})$ is able to break opaque predicate $\forall x \in \mathbb{Z} : f(x) \subseteq P$.

5.3.2 Experimental results

A prototype of the above described attack based on the abstract domain *Parity* has been implemented using Loco [105], a x86 tool for obfuscation/deobfuscation transformations which is able to insert opaque predicates. This experimental evaluation has been conducted on the aforementioned 1.6 GHz Pentium M-based system. Each program of the SPECint2000 benchmark suite is obfuscated by inserting the following true opaque predicates: $\forall x \in \mathbb{Z} : 2|(x^2 + x)$ and $\forall x \in \mathbb{Z} : 2|(x + x)$. It turns out that *Parity* is $\mathcal{B}$-complete for addition, square and identity function, thus by Corollary 5.9, the abstract domain *Parity* models an attacker that is able to break these opaque predicates. In the obfuscating transformation each basic block of the input assembly program is split into two basic blocks. Then, Loco checks whether the opaque predicate can be inserted between these two basic blocks: a liveness analysis is used here to ensure that no dependency is broken and that the obfuscated program is functionally equivalent to the original one. In particular, liveness analysis checks that the registers and the conditional flags affected by the opaque predicate are not live in the program point where the opaque predicate will be inserted. Moreover, our tool checks by a standard constant propagation whether the registers associated to the opaque predicate are constant or not. If constant propagation detects that these are constant then the opaque predicate can be trivially broken and therefore is not inserted. Although liveness analysis and constant propagation are noticeably time-consuming, they are nevertheless necessary both to ensure functional equivalence between original and obfuscated program and to guarantee that the opaque predicate cannot be trivially broken by constant propagation. The algorithm used to detect opaque predicates is analogous to the brute force attack algorithm described in Section 5.2.2. Fig. 5.7 describes the basic block, by pseudo-code, which implements the opaque predicate $\forall x \in \mathbb{Z} : 2|(x^2 + x)$.

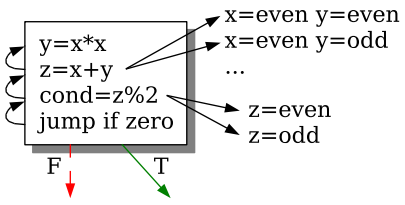


Fig. 5.7. Breaking $\forall x \in \mathbb{Z},\ 2|(x^2 + x)$

Let us describe how our deobfuscation algorithm works. For each conditional jump j, `jump if zero` in the figure, we consider the instruction i which immediately precedes j, `cond=z%2` in the figure. The instructions j and i are abstractly executed on each value of the abstract domain (i.e., the attack). In the considered case of the attack modeled by *Parity*, both non-trivial values *even* and *odd* are given as input to `cond=z%2`. When `z` evaluates to *even*, `cond` evaluates to 0 and therefore the true path is followed. On the other hand, when `z` is evaluated to *odd*, `cond` evaluates to 1 and the false path is taken. Thus, i does not give rise to an opaque predicate, so that we need to consider the instruction `z=x+y` which immediately precedes i. The instruction `z=x+y` is binary and therefore we need to consider all the values in *Parity* $\times$ *Parity*. This process is iterated until an opaque predicate is detected or the end of the basic block is reached. In our case, the opaque predicate is detected when the algorithm analyses the instruction `y=x*x` because whether `x` is evaluated to *even* or *odd* the true path is taken. The number of computational steps needed for breaking one single opaque predicate by an attack based on an abstract domain A is $n^2 * d^r$, where n is the number of instructions composing the opaque predicate, r is the number of registers used by the opaque predicate and d is the number of abstract values in A. The reduction of the computational effort of the abstract interpretation-based attack with respect to the brute force attack can therefore be huge since the abstract domain can encode a very coarse approximation (namely d may be much smaller than 2^w where w is the register width). Since in the considered example the opaque predicate consists of 3 instructions, uses 2 registers and *Parity* has 2 non-trivial abstract values, the number of steps for detecting $\forall x \in \mathbb{Z} : 2|x + x$ through the abstract domain *Parity* becomes $3^2 * 2^2$.

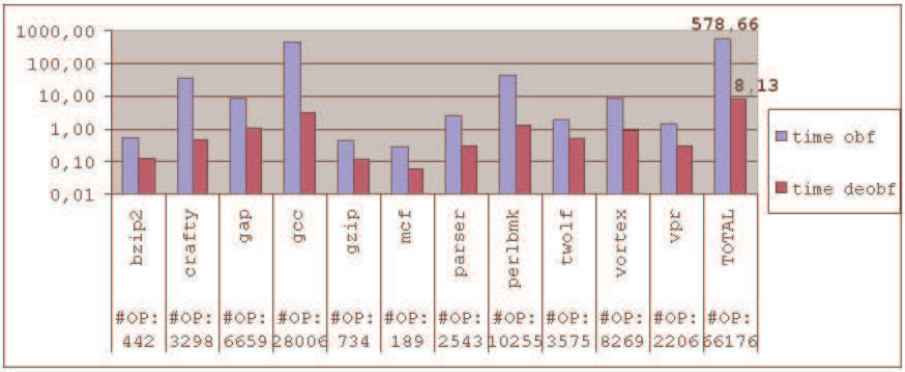


Table 5.3. Timings of obfuscation and deobfuscation

In Table 5.3 we show the results of the obfuscation/deobfuscation process on the standard suite of benchmarks SPECint2000. For each program we report

the number $\sharp OP$ of inserted opaque predicates and the time needed to obfuscate and deobfuscate the program, that is the time needed to insert and to detect the considered opaque predicates. For each program the left (blue) column represents the time (in seconds) needed to insert the opaque predicates and the right (violet) column represents the time needed to detect the inserted opaque predicates. It turns out that the *Parity*-based deobfuscation process is able to detect all the inserted opaque predicates. Let us recall that the brute force attack took 8.83 seconds to detect only one occurrence of the opaque predicate $\forall x \in \mathbb{Z} : 2|x+x$ in a 16-bit environment, while the abstract interpretation-based deobfuscation attack took 8.13 seconds to deobfuscate 66176 opaque predicates in a 32-bit environment. Observe that, in general, the time needed to obfuscate is grater than the time needed to deobfuscate, due to the fact that the insertion of opaque predicates needs some preliminary static analysis which can be time consuming.

The experimental results show the improvement in performance obtained from the theoretical investigation. It is clear that the approach described for this class of opaque predicates can be applied to other classes of predicates. As an example, in the next section we consider another class of numerical opaque predicates and show that, once again, predicate detection can be reduced to a completeness property of abstract domains.

5.3.3 Breaking Opaque Predicates $h(x) = g(x)$

In [35] Collberg et al. observe that the study of random Java programs reveals that most predicates are extremely simple. In particular, common patterns include the comparison of integer quantities using binary operators such as equal to, greater than, smaller than, etc. It is clear that, in order to design stealthy obfuscating transformations, the inserted opaque predicates have to resemble the structure of predicates typically present in a program. For this reason we restrict our study to numerical opaque predicates on integer values. In general, an opaque predicate of this kind is a function $\mathbb{Z}^n \to \mathcal{B}_\perp$ that takes an array of n integer values and returns *true*, *false*, $\perp$ or $\top$. A wide class of numerical opaque predicates can be characterized by the following structure:

$$\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) \ \mathbf{compare} \ g(\bar{x})$$

where **compare** stands for any binary operator in the set $\{=, \geq, \leq\}$, $\bar{x}$ is an array of n integer values, namely $\bar{x} \in \mathbb{Z}^n$, h and g are two functions over integers, in particular $h, g : \mathbb{Z}^n \to \mathbb{Z}$. Let us assume that each variable of program P ranges over $\mathbb{Z}$ and let $|var[\![P]\!]| = m$. Each abstract domain (attacker) $\varphi \in uco(\wp(\mathbb{Z}^m))$ induces an abstraction on the values of variables and therefore on the value that the opaque predicate input can assume. From now on, the abstract domain

$\varphi \in uco(\wp(\mathbb{Z}^n))$ models the attacker that observes an approximation φ of opaque predicate inputs. Let us consider a numerical opaque predicate of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, which verifies whether two functions h and g always return the same value when applied to the same array of integer values. In order to precisely detect the opaqueness of $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, one needs to check the *concrete test*, denoted as $\mathrm{CT}^{h,g}$ and defined as follows:

$$\mathrm{CT}^{h,g} \stackrel{\text{def}}{=} \forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$$

Once again, the set of predicates that satisfy the concrete test corresponds to the set OP of predicates characterizes by (5.1). Our goal is to characterize the family of abstractions of $\wp(\mathbb{Z}^n)$ that perform the test of opaqueness for h and g in a precise way, namely the set of abstractions that loose information that is irrelevant for the precise computation of h and g. We are therefore interested in the family of abstract domains that are able to precisely compute functions h and g, which corresponds to the class of attackers able to deobfuscate the insertion of predicates of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$. Given a set $X \subseteq \mathbb{Z}^n$ let $h(X) \doteq g(X)$ denote the point to point definition of equality, where $h(X) \doteq g(X)$ if and only if $\forall \bar{x} \in X : h(\bar{x}) = g(\bar{x})$. Let $\mathrm{AT}^{h,g}_{\varphi}$ denote the *abstract test* for opaqueness associated to an attacker modeled by the abstract domain φ. The abstract test is defined as follows:

$$\mathrm{AT}^{h,g}_{\varphi} \stackrel{\text{def}}{=} \forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\bar{x}))) \doteq \varphi(g(\varphi(\bar{x})))$$

Also in this case the set of opaque predicates satisfying the abstract test on φ corresponds to the set OP^{φ} of opaque predicates characterized by (5.2). Once again, the precision of the abstract test strongly depends on the considered abstract domain. Thus, as in Section 5.3.1, sound and complete abstract tests are defined as follows.

Definition 5.14. Given an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, and an abstraction $\varphi \in uco(\wp(\Sigma^+))$, we say that:

- $\mathrm{AT}^{h,g}_{\varphi}$ is *sound* when $\mathrm{AT}^{h,g}_{\varphi} \Rightarrow \mathrm{CT}^{h,g}$
- $\mathrm{AT}^{h,g}_{\varphi}$ is *complete* when $\mathrm{CT}^{h,g} \Rightarrow \mathrm{AT}^{h,g}_{\varphi}$

When the abstract test $\mathrm{AT}^{h,g}_{\varphi}$ is both sound and complete, i.e., $\mathrm{AT}^{h,g}_{\varphi} \Leftrightarrow \mathrm{CT}^{h,g}$, we say that attacker φ *breaks* the opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$. In fact, in this case the set of opaque predicates coincides with the set of opaque predicates classified as opaque by the abstract test, meaning that we have obtained the desired equality $OP = OP^{\varphi}$.

It turns out that, considering opaque predicates of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, for any abstract domain $\varphi \in uco(\wp(\mathbb{Z}^n))$ modeling the attacker, the abstract test defined above is always complete.

Corollary 5.15. $\mathrm{AT}^{g,h}_{\varphi}$ is complete.

PROOF: If the concrete test $\mathrm{CT}^{h,g}$ is verified we have that $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, since $\varphi(\bar{x}) \subseteq \mathbb{Z}^n$ then $\forall \bar{x} \in \mathbb{Z}^n : \forall \bar{y} \in \varphi(\bar{x}) : h(\bar{y}) = g(\bar{y})$. This means that $\forall \bar{x} \in \mathbb{Z}^n : h(\varphi(\bar{x})) \doteq g(\varphi(\bar{x}))$, thus $\forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\bar{x}))) \doteq \varphi(g(\varphi(\bar{x})))$ that corresponds to the satisfaction of the abstract test $A^{g,h}_{\varphi}$.

□

This means that if a predicate is opaque then the attacker recognises it, namely $OP \subseteq OP^{\varphi}$. Thus, $d_{OP^{\varphi}}(\mathbf{S}^{\varphi}[\![P]\!]) = d_{OP^{\varphi}}(\mathbf{S}^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$. In fact, $d_{OP^{\varphi}}$ eliminates all the opaque predicates from the right term and the common regular predicate that are erroneously classified as opaque from both terms. For the same reason we have that $\mathbf{S}^{\varphi}[\![P]\!] \neq d_{OP^{\varphi}}(\mathbf{S}^{\varphi}[\![P]\!])$. This means that $\mathbf{S}^{\varphi}[\![P]\!] \neq d_{OP^{\varphi}}(\mathbf{S}^{\varphi}[\![\mathbb{t}^{OP}[\![P, \mathfrak{I}[\![P]\!]]\!]]\!])$ and therefore that attacker φ is defeated. As argued above, attacker φ is able to break opaque predicates insertion when $OP = OP^{\varphi}$, which is guaranteed when the abstract test $\mathrm{AT}^{h,g}_{\varphi}$ is both sound and complete. Corollary 5.15 guarantees completeness of the abstract test, thus, in order to break an opaque predicate, we need to verify the soundness condition. In general $\mathrm{AT}^{h,g}_{\varphi}$ is not sound, but it is possible to show that soundness is guaranteed when the abstract domain φ modeling the attacker is $\mathcal{F}$-complete for both functions h and g.

Theorem 5.16. Given an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, and an attacker modeled by $\varphi \in uco(\wp(\mathbb{Z}^n))$, if the abstraction φ is $\mathcal{F}$-complete for both functions h and g then the abstract test $\mathrm{AT}^{h,g}_{\varphi}$ is sound.

PROOF: We have to prove that $\mathrm{AT}^{h,g}_{\varphi} \Rightarrow \mathrm{CT}^{h,g}$. If the abstract test $\mathrm{AT}^{h,g}_{\varphi}$ holds then $\forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\bar{x}))) \doteq \varphi(g(\varphi(\bar{x})))$, namely $\forall \bar{x} \in \mathbb{Z}^n : \varphi(h(\varphi(\varphi(\bar{x})))) \doteq \varphi(g(\varphi(\varphi(\bar{x}))))$. The abstract domain φ is $\mathcal{F}$-complete by hypothesis, therefore $\forall \bar{x} \in \mathbb{Z}^n : h(\varphi(\varphi(\bar{x}))) \doteq g(\varphi(\varphi(\bar{x})))$, which is equivalent to $\forall \bar{x} \in \mathbb{Z}^n : h(\varphi(\bar{x})) \doteq g(\varphi(\bar{x}))$. By definition of $\doteq$ this means that $\forall \bar{x} \in \mathbb{Z}^n : \forall \bar{y} \in \varphi(\bar{x}) : h(\bar{y}) = g(\bar{y})$. φ is extensive by hypothesis, namely $\bar{x} \in \varphi(\bar{x})$, and therefore $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, that corresponds to the satisfaction of the concrete test $\mathrm{CT}^{h,g}$.

□

This means that, when the abstract domain modeling the attacker is able to precisely compute the functions composing the opaque predicate, then the attacker breaks the opaque predicate. Thus, given an attacker φ and an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$, the $\mathcal{F}$-completeness domain refinement of φ with respect to functions h and g adds the minimal amount of information to attacker φ to make it able to defeat the considered opaque predicate. Hence, completeness domain refinement provides here a systematic technique to design attackers that are able to break an opaque predicate of interest. Once again, the

completeness property of abstract interpretation precisely captures the ability of an attacker to disclose an opaque predicate.

The above result holds also when considering $\leq, \geq$, and the corresponding point to point extensions $\dot{\leq}, \dot{\geq}$, instead of $=$ and $\doteq$.

Corollary 5.17. Given an opaque predicate $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x})$ **compare** $g(\bar{x})$, and an attacker modeled by $\varphi \in uco(\wp(\mathbb{Z}^n))$, if the abstraction φ is $\mathcal{F}$-complete for both functions h and g, then φ breaks opaque predicates that are instances of $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x})$ **compare** $g(\bar{x})$.

In the following example we show how the lack of $\mathcal{F}$-completeness of the abstract domain modeling the attacker can cause the abstract test to hold, even if the concrete one fails.

Example 5.18. Let us consider the predicate $\forall x \in \mathbb{Z} : 2x^2 = 2x$, where $h(x) = 2x^2$ and $g(x) = 2x$. It is clear that $\mathrm{CT}^{h,g}$ does not hold, since the predicate is not opaque. Let us consider an attacker modeled by the abstract domain of $Parity = \{\top, \bot, even, odd\}$. In turns out that $\mathrm{AT}^{h,g}_{Parity}$ holds, in fact:

$$even :: Parity(h(even)) = even = Parity(g(even))$$
$$odd :: Parity(h(odd)) = even = Parity(g(odd))$$

The reason why the abstract test holds on *Parity* is the fact that *Parity* is not $\mathcal{F}$-complete for both h and g. In fact, let $Parity = \gamma \circ \alpha$, then $2(\gamma(even)) = \left\{ 2x \,\middle|\, x \in 2\mathbb{Z} \right\}$ which is strictly contained in $\gamma(2even) = \gamma(even) = 2\mathbb{Z}$. When computing the $\mathcal{F}$-completeness domain refinement of *Parity* with respect to h and g, we close the considered abstract domain with respect to h and g. This means that, for example the elements $Double_2$, such that $\gamma(Double_2) = \left\{ 2x \,\middle|\, x \in 2\mathbb{Z} \right\}$, $Double_1$, such that $\gamma(Double_1) = \{2x \mid x \in 2\mathbb{Z} + 1\}$, $DoubleSq_2$, such that $\gamma(DoubleSq_2) = \{2x^2 \mid x \in 2\mathbb{Z}\}$, and $DoubleSq_1$, such that $\gamma(DoubleSq_1) = \{2x^2 \mid x \in 2\mathbb{Z}+1\}$, belong to $\mathcal{R}^{\mathcal{F}}_{h,g}(Parity) = Parity^+$. Observe that on this domain the abstract test does not hold any more, in fact $Parity^+(h(even)) = DoubleSq_2 \neq Double_2 = Parity^+(g(even))$, and so on for all the other elements since the direct image of all elements under h and g are precisely expressed by the domain obtained through the completeness refinement.

□

Observe that the theoretical investigation of Section 5.3.1 takes into account the elementary functions implementing the opaque predicates while in this section we do not consider such details. It is clear that, in order to provide some experimental results also for the class of opaque predicates of the form $\forall \bar{x} \in \mathbb{Z}^n : h(\bar{x}) = g(\bar{x})$ the $\mathcal{F}$-completeness problem needs to be described in terms of elementary functions composing the predicates (which can be easily done since completeness is preserved by composition).

5.3.4 Comparing Attackers

The completeness results obtained in Section 5.3.1 and in Section 5.3.3 allow us to compare on the lattice of abstract interpretation both the efficiency of different attackers in disclosing a particular opaque predicate, and the resilience of different opaque predicates with respect to an attacker.

Let P^T denote a predicate belonging to either one of the two considered classes of opaque predicates, and let us denote with $\mathcal{R}_{P^T}$ the completeness domain refinement needed to make an attacker able to break P^T (without distinguishing between backward or forward completeness refinements). Let $Potency(P^T, \varphi)$ denote the potency of opaque predicate P^T in contrasting attacker φ, and $Resilience(P^T, \varphi)$ the resilience of opaque predicate P^T in preventing attacker φ.

Definition 5.19. Given two attackers $\varphi, \phi \in uco(\wp(\Sigma^+))$ and two opaque predicates P_1^T and P_2^T:

- if $\varphi \sqsubset \psi$ and $\mathcal{R}_{P^T}(\psi) = \mathcal{R}_{P^T}(\varphi)$, we say that $Potency(P^T, \psi)$ is greater than $Potency(P^T, \varphi)$
- when $\mathcal{R}_{P_1^T}(\varphi) \sqsubset \mathcal{R}_{P_2^T}(\varphi)$, we say that $Resilience(P_1^T, \varphi)$ is greater than $Resilience(P_2^T, \varphi)$

The first point of the above definition refers to the situation represented in Fig. 5.8 (a), where $\varphi \sqsubset \psi$ and $\mathcal{R}_{P^T}(\psi) = \mathcal{R}_{P^T}(\varphi)$. In this case we have that the insertion of P^T contrasts attacker ψ more than what contrasts attacker φ. This is because more information needs to be added to ψ than to φ in order to gain an attacker able to break P^T, namely ψ is farther away than φ from disclosing P^T.

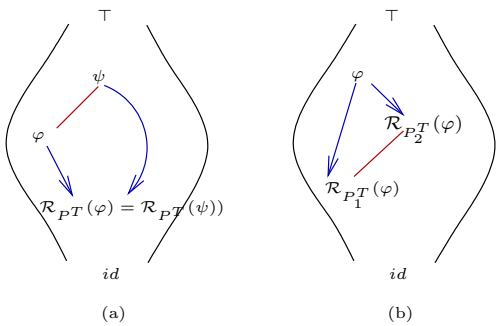


Fig. 5.8. Comparing attackers

The same reasoning allows us to compare the resilience of different opaque predicates in the lattice of abstract interpretation. In fact, the second point

of the above definition considers two predicates P_1^T and P_2^T and an attacker $\varphi \in uco(\wp(\Sigma^+))$ and assumes that $\mathcal{R}_{P_1^T}(\varphi) \sqsubset \mathcal{R}_{P_2^T}(\varphi)$ as shown in Fig. 5.8 (b). In this case we can say that the insertion of opaque predicate P_1^T is more efficient in obstructing attacker φ than the insertion of opaque predicate P_2^T, because more information needs to be added to φ in order to disclose P_1^T than P_2^T. Thus, in order to understand which opaque predicate in OP is more efficient for contrasting a given attacker φ, it is necessary to compute the fixpoint solution of the completeness domain refinement of φ with respect to the different opaque predicates available, and then choose the one that corresponds to the most concrete refinement. In fact, the closer the refined attacker is to the identical abstraction (concrete semantics), the higher is the resilience of the opaque predicate. In particular, if $\mathcal{R}_{P^T}(\varphi) = id$, it means that the attacker φ can break the considered opaque predicate only if it can access the concrete program semantics. In this case the considered opaque predicate provides the best obstruction to φ.

5.4 Discussion

We have studied the effects that opaque predicate insertion has on program trace semantics and we have systematically derived a possible obfuscating algorithm following the methodology proposed by Cousot and Cousot [44]. The semantic understanding of opaque predicate insertion leads us to observe how this transformation does not irremediably affect program trace semantics. As usual, we assume that attackers have a constrained observation of a program's behaviour, and this is specified by modeling attackers as abstractions of trace semantics. The semantics-based notion of potency given in Definition 4.3 states that a transformation $\mathbb{t}$ is potent if it defeats attackers modeled as properties of program trace semantics, namely if there exists a property φ such that $\varphi(\mathbf{S}^+[\![P]\!]) \neq \varphi(\mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!])$. This measure of potency fits transformations that deeply modify program trace semantics, and provides an advanced technique for comparing obfuscating algorithms relatively to their potency in the lattice of abstract interpretation (as stated by Definition 4.10). However, Definition 4.3 is not adequate for modeling the potency of obfuscating transformations that leave program trace semantics almost unchanged, as in the case of opaque predicate insertion. In this case, we need a notion of program potency that captures the *noise* introduced at the level of program control flow, which is an abstraction of trace semantics. This observation has led to Definition 5.5, where transformation potency is formalized with respect to the abstract semantics computed on the abstract domain modeling the attacker, namely a transformation $\mathbb{t}$ is potent if there exists an abstraction φ such that $\mathbf{S}^{\varphi}[\![P]\!] \neq \mathbf{S}^{\varphi}[\![\mathbb{t}[\![P]\!]]\!]$. It is clear how the two definitions of potency are deeply different and orthogonal and how each of them fits different kinds of obfuscations. In Chapter 6 we classify program

transformations according to the effects that they have on trace semantics. In particular, a transformation $\mathbb{t}$ is conservative when the semantics of the original and obfuscated program share the same structure (more formally when for each trace $\sigma \in \mathbf{S}^+[\![P]\!]$ there exists a trace $\delta \in \mathbf{S}^+[\![\mathbb{t}[\![P]\!]]\!]$ that presents all the states of σ in the same order), non-conservative otherwise. In Chapter 6 we discuss the importance of this classification. This classification turns out to be related to the above-mentioned definitions of potency, in fact Definition 4.3 of transformation potency suites non-conservative obfuscations, while Definition 5.5 suites conservative obfuscations.

In the particular case of opaque predicate insertion, the use of abstract interpretation ensures that, when the abstraction is complete, the attacker is able to break the opaque predicate, namely to remove the obfuscation. This proves that deobfuscation in the case of opaque predicates requires complete abstractions and therefore the potency of opaque predicates can be measured by the amount of information that has to be added to the incomplete domain to become complete. This allows us to compare both the potency of different opaque predicates with respect to a given attack, and the resilience of an opaque predicate with respect to different attackers. Some further work is necessary in order to validate our theory in practice. In fact, while measuring the resilience of opaque predicates in the lattice of abstract domains may provide an absolute and domain-theoretical taxonomy of attackers and obfuscators, it would be interesting to investigate the true effort, in terms of dynamic testing, which is necessary to enforce static analysis in order to break opaque predicates. We believe that this is proportional to the missing information in the abstraction modeling the static analysis with respect to its complete refinement. However, preliminary works on this directions show promising experimental results, as described in Section 5.3.2.

As observed above, the insertion of an opaque predicate creates a path that is never taken. It is clear that when the false path of a true opaque predicate contains another opaque predicate the degree of obfuscation of the transformation increases. The two opaque predicates interact with each other, and this dependence adds more confusion in the understanding of the original control flow of the program. Thus, we propose the insertion of dependent opaque predicates as a new and more potent obfuscation technique.

Consider for example the true opaque predicates $P1 : \forall x \in \mathbb{Z} : 2|(x^2 + x)$ and $P2 : \forall x \in \mathbb{Z} : 3|(x^3 - x)$ that interact with each other as depicted Fig 5.9. On the left-hand side we have the opaque predicate $P1$, while on the right-hand side we have $P2$, expressed in terms of elementary functions, i.e., assembly instructions. Observe that the false branch of predicate $P1$ enters the second basic block of predicate $P2$ and vice versa. Following our completeness result, the attacker modeled by the abstract domain *Parity* should be able to break opaque predicate $P1$. The problem is that *Parity* cannot break $P2$ and therefore

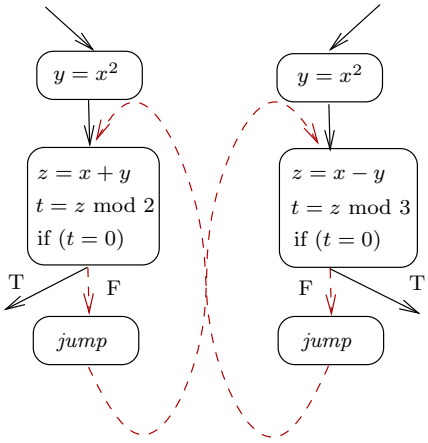


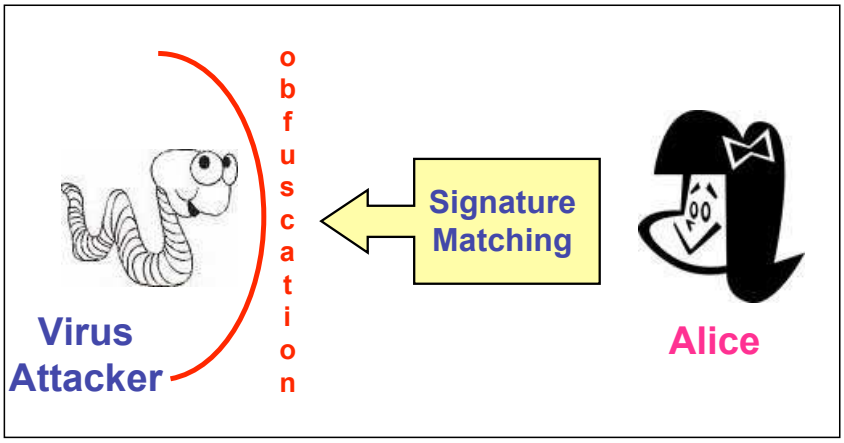
Fig. 5.9. Dependent opaque predicates

we have an incoming edge on the second basic block of opaque predicate $P1$ coming from $P2$. This gives the idea of why we are no longer able to break opaque predicate $P1$ with the *Parity* domain. Therefore, when there are opaque predicates that interact with each other the attacker needs to take into account these dependencies. Our guess is that a suitable attacker to handle this situation could probably be obtained by combining the abstract domains breaking the individual opaque predicates. This means that one opaque predicate which is not breakable by our technique could protect breakable opaque predicates by interacting with them.

Another aspect that we would like to investigate is the use of abstract domains that are more complex than the ones considered so far in order to construct new opaque predicates and to detect more sophisticated ones. The idea is that program properties that can be studied only on complex domains could lead to the design of novel opaque predicates. Since these properties derive from a complex analysis the corresponding opaque predicates should be resilient to attacks. Consider for example the polyhedral abstract domain [45] and the abstract domain of octagons [113] for discovering properties of numerical variables.

6

A Semantics-Based approach to Malware Detection



The Malicious Code Perspective

The theoretical framework proposed in Chapter 4 and Chapter 5 provides a formal setting where to understand code obfuscation from a semantic point of view. As noticed earlier, potency of obfuscating transformations and attackers, namely users interested in recovering the original code, can both be modeled as abstractions of program trace semantics. This allows us to compare potency and resilience of different obfuscating transformations with respect to different attackers on the lattice of abstract interpretation. It is clear that the results of the previous chapters still hold when considering obfuscating transformations as malicious transformations used by malware writers to prevent detection, and de-obfuscation tools, i.e., attackers, as malware detection algorithms. Notice that, if in the software protection field we are interested in the design of resilient ob-

fuscating techniques that are able to contrast as many attacks as possible, in the malware detection scenario, we are interested in defeating as many obfuscations as possible.

As observed in Section 3.2, a *malware* is a program with a malicious intent that has the potential to harm the machine on which it executes or the network over which it communicates. A *malware detector* is a tool designed to identify malware and the design of efficient malware detection schemes is a crucial aspect of software security. As argued earlier, a *misuse malware detector* (or, alternately, a *signature-based malware detector*) is based on a list of signatures (traditionally known as a *signature database* [114]). The idea is that, when part of a program matches a signature in the database, the program is classified as infected by the malware [140]. Misuse malware detectors' low false-positive rate and ease of use have led to their widespread deployment. Other approaches for identifying malware have not proved practical as they suffer from high false positive rates (e.g., anomaly detection using statistical methods [87,96]) or can only provide a post-infection forensic capability (e.g., correlation of network events to detect propagation after infection [68]). Malware writers continuously test the limits of malware detectors in an attempt to discover ways to evade detection. This leads to an ongoing game of one-upmanship [119], where malware writers find new ways to create undetected malware, and where researchers design new signature-based techniques for detecting such evasive malware. This co-evolution is a result of the theoretical undecidability of malware detection [20, 27]. This means that, in the currently accepted model of computation, no ideal malware detector exists. The only achievable goal in this scenario is to design better detection techniques that jump ahead of evasion techniques and make the malware writers task harder.

We have already observed how code obfuscation can be used to foil malware detection algorithms based on signature matching, which attempt to capture (syntactic) characteristics of the machine-level byte sequence of the malware. This reliance on a syntactic approach makes such detectors vulnerable to code obfuscation that alters syntactic properties of the malware byte sequence without significantly affecting their execution behavior. If a signature describes a certain sequence of instructions [140], then those instructions can be reordered or replaced with equivalent instructions [155, 156]. Such obfuscations are especially applicable on CISC architectures, such as the Intel IA-32 [76], where the instruction set is rich and many instructions have overlapping semantics. If a signature describes a certain distribution of instructions in the program, insertion of junk code [80, 141, 156] that acts as a NOP so as not to modify the program behavior can defeat frequency-based signatures. If a signature identifies some of the read-only data of a program, packing or encryption with varying keys [52, 129] can effectively hide the relevant data. Therefore, an important

requirement of a robust malware detection technique is to handle obfuscating transformations.

In this chapter we take the position that *the key to identify (possibly obfuscated) malware lies in a deeper investigation of their semantics*. Program semantics provides a formal model of program behavior, therefore addressing the malware-detection problem from a semantic point of view could lead to a more robust detection system.

We propose a semantics-based framework for reasoning about malware detectors and proving properties such as soundness and completeness of these detectors. The basic idea of our approach is to use trace semantics to characterize the behaviors of the malware as well as of the program to be checked for infection, and to use abstract interpretation to "hide" irrelevant aspects of these behaviors. Preliminary work by Christodorescu et al. [25] and Kinder et al. [84] on a formal approach to malware detection confirms the potential benefits of a semantics-based approach. Moreover, the proposed semantics-based framework can be used by security researchers to reason about and evaluate (prove) the resilience of malware detectors to various kinds of obfuscating transformations. In particular, we present a formal definition of what it means for a detector to be sound (i.e., no false positives) and complete (i.e., no false negatives) with respect to a class of obfuscations, together with a formal framework that malware-detection researchers can use to prove completeness and soundness of their algorithms with respect to classes of obfuscations. As an integral part of the formal framework, we provide a trace semantics to characterize the program and malware behaviors. In Section 6.6, we investigate the relation between the semantics-based malware detector and the signature matching algorithm and we prove that signature matching approaches are generally sound, while they are complete only for a restricted class of obfuscating transformations. Moreover, in Section 6.7, we show our formal framework in action by proving that the semantics-aware malware detector $\mathcal{A}_{MD}$ proposed by Christodorescu et al. [25] is complete with respect to some common obfuscations used by malware writers. The soundness of $\mathcal{A}_{MD}$ was proved in [25]. The results presented in this chapter have been published in [46].

6.1 Overview

In this section we provide definitions of what it means for a malware detector to be sound and complete with respect to a class of obfuscations, together with the description of a possible strategy to prove such properties in a semantics-based framework (Section 6.1.1). In Section 6.1.2, we introduce the syntax and the semantics of the programming language used in this chapter.

As usual, an obfuscating transformation, denoted as $\mathcal{O} : \mathbb{P} \to \mathbb{P}$, is a potent program transformer that preserves program functionality to some extent. Let $\mathbb{O}$ denote the set of all obfuscating transformations. A malware detector can be seen as a function $D : \mathbb{P} \times \mathbb{P} \to \{0, 1\}$ that, given a program P and a malware M, decides if program P is infected by malware M. For example, $D(P, M) = 1$ means that program P is infected with malware M or with an obfuscated variant $\mathcal{O}[\![M]\!]$ where $\mathcal{O} \in \mathbb{O}$. Our treatment of malware detectors is focused on detecting variants of existing malware. When a program P is infected with a malware M, we write $M \hookrightarrow P$. The precision of a malware detector can be formalized in terms of soundness and completeness properties. Intuitively, a malware detector is *sound* if it never erroneously claims that a program is infected, i.e., there are no *false positives*, and it is *complete* if it always detects programs that are infected, i.e., there are no *false negatives*. More formally, these properties can be defined as follows.

Definition 6.1.

– A malware detector D is *complete* for an obfuscation $\mathcal{O} \in \mathbb{O}$ if and only if

$$\forall M, P \in \mathbb{P} : \quad \mathcal{O}[\![M]\!] \hookrightarrow P \Rightarrow D(P, M) = 1$$

– A malware detector D is *sound* for an obfuscation $\mathcal{O} \in \mathbb{O}$ if and only if

$$\forall M, P \in \mathbb{P} : \quad D(P, M) = 1 \Rightarrow \mathcal{O}[\![M]\!] \hookrightarrow P$$

Observe that, the aim of an attacker observing a program that uses code obfuscation to protect its sensitive information, is to recover enough information on the original program in order to perform reverse engineering. On the other side, the goal of malware detection is to understand if a certain program is an obfuscated version of another one, with no need of recovering the original malware. Besides this difference, the proposed definitions of soundness and completeness can be applied to deobfuscating techniques as well. In other words, our definitions are not tied to the concept of malware detection.

Most malware detectors are built on top of other static-analysis techniques for problems that are hard or undecidable. For example, most malware detectors [25, 84] that are based on static analysis assume that the control flow graph for an executable can be extracted. As shown by researchers [100], simply disassembling an executable can be quite tricky. Therefore, we want to introduce the notion of *relative soundness and completeness* with respect to algorithms that a detector uses. In other words, we want to prove that a malware detector is sound or complete with respect to a class of obfuscations if the static analysis algorithms that the detector uses are perfect. This allows us to measure the precision of a given detection algorithm independently from the precision of related static analysis algorithms.

Definition 6.2. An *oracle* is an algorithm over programs that provides perfect answers in time $O(1)$. For example, a *CFG* oracle is an algorithm that takes a program as an input and produces its control flow graph.

Let $D^{\mathcal{OR}}$ denote a malware detector that uses a set of oracles $\mathcal{OR}$[1]. For example, let OR_{CFG} be a static analysis oracle that given an executable provides a perfect control flow graph for it. Thus, a detector that uses the oracle OR_{CFG} is denoted $D^{OR_{CFG}}$. In the following, when proving soundness and completeness of a given malware detector in the semantics-based framework, we will assume that the oracles that the detector uses are perfect. Soundness (resp. completeness) with respect to perfect oracles is also called *oracle soundness* (resp. *oracle completeness*).

Definition 6.3. A malware detector $D^{\mathcal{OR}}$ is *oracle complete* with respect to an obfuscation $\mathcal{O}$, if $D^{\mathcal{OR}}$ is complete for that obfuscation $\mathcal{O}$ when all oracles in the set $\mathcal{OR}$ are perfect. *Oracle soundness* of a detector $D^{\mathcal{OR}}$ can be defined in a similar manner.

6.1.1 Proving Soundness and Completeness of Malware Detectors

When a new malware detection algorithm is proposed, one of the criteria of evaluation is its resilience to obfuscations, both current and future. In fact, when an attacker, i.e., a malware writer, has access to the detection algorithm and to its inner workings, he can use such knowledge in order to design ad-hoc obfuscation tools to bypass such detection scheme. As the malware detection problem is in general undecidable, it is always possible to design a new obfuscating transformation that defeats a given detector. Unfortunately, identifying the classes of obfuscations for which a detector is resilient can be a complex and error-prone task. A large number of obfuscation schemes exist, both from the malware world and from the intellectual property protection industry. Furthermore, obfuscations and detectors are defined using different languages (e.g., program transformation vs program analysis), complicating the task of comparing one against the other.

In the following, we present a formal framework for proving soundness and completeness of malware detectors in the presence of obfuscating transformations. This framework operates on programs described through the collection of their execution traces – thus, program trace semantics is the building block of our approach. In particular, in Section 6.2 and Section 6.3, we describe how both obfuscations and detectors can be elegantly expressed as operations on traces, and in Section 6.4 we characterize classes of obfuscating transformations

[1] We assume that detector D can query an oracle from the set $\mathcal{OR}$, and the query is answered perfectly and in $O(1)$ time. These types of relative completeness and soundness results are common in cryptography.

in terms of the effects that they have on program trace semantics, and we prove soundness and completeness of malware detectors with respect to such classes of transformations. Our approach allows us to certify that a certain detection algorithm is able to deal with all obfuscations (even future ones) that satisfy a certain property.

In this formal setting, we propose the following two step *proof strategy* for showing that a detector $D^{\mathcal{OR}}$ is sound or complete with respect to an obfuscation or a class of obfuscations.

Step 1: Relating the two worlds.
Consider a malware detector $D^{\mathcal{OR}}$ that uses a set of oracles $\mathcal{OR}$. Given a program P and malware M, let $\mathbf{S}[\![P]\!]$ and $\mathbf{S}[\![M]\!]$ denote the set of traces corresponding to the semantics of P and M respectively. In Section 6.2 and Section 6.3 we describe a detector D_{Tr} which works in the semantic world of traces, and classifies a program P as infected by a malware M if the semantics of P matches the semantics of M up to abstraction α (where the matching relation up to α will be precisely defined later). Thus, the first step is to prove that, given a proper abstraction α and assuming that the oracles in $\mathcal{OR}$ are perfect, the two detectors are equivalent, i.e., for all P and M in $\mathbb{P}$: $D^{\mathcal{OR}}(P, M) = 1$ if and only if $D_{Tr}(\alpha(\mathbf{S}[\![P]\!]), \alpha(\mathbf{S}[\![M]\!])) = 1$. In other words, this step shows the equivalence of the two worlds: the concrete world of programs and the semantic world of traces.

Step 2: Proving soundness and completeness in the semantic world.
After step 1, we are ready to prove the desired property (e.g., completeness) about the trace-based detector D_{Tr} on α, with respect to the chosen class of obfuscations. In this step, the detector's effects on trace semantics are compared to the effects of obfuscations on trace semantics. This allows us to evaluate the detector against whole classes of obfuscations, as long as the obfuscations have similar effects on trace semantics.

The requirement for equivalence in step 1 above might be too strong if only one of completeness or soundness is desired. For example, if the goal is to prove only completeness of a malware detector $D^{\mathcal{OR}}$, then it is sufficient to find a trace-based detector that classifies only malware and malware variants in the same way as $D^{\mathcal{OR}}$. Then, if the trace-based detector is complete, so is $D^{\mathcal{OR}}$.

Observe that the proof strategy presented above works under the assumption that the set of oracles $\mathcal{OR}$ used by the detector $D^{\mathcal{OR}}$ are perfect. In fact, the equivalence of the semantic malware detector D_{Tr} to the detection algorithm $D^{\mathcal{OR}}$ is stated and proved under the hypothesis of perfect oracles. This means that when the oracles in $\mathcal{OR}$ are perfect then:

- $D^{\mathcal{OR}}$ is sound with respect to obfuscation $\mathcal{O} \Leftrightarrow D_{Tr}$ is sound with respect to obfuscation $\mathcal{O}$
- $D^{\mathcal{OR}}$ is complete with respect to obfuscation $\mathcal{O} \Leftrightarrow D_{Tr}$ is complete with respect to obfuscation $\mathcal{O}$

Consequently, the proof of soundness (resp. completeness) of D_{Tr} with respect to a given obfuscation $\mathcal{O}$ implies soundness (resp. completeness) of $D^{\mathcal{OR}}$ with respect to obfuscation $\mathcal{O}$ and viceversa. However, even when the oracles used by the detection scheme $D^{\mathcal{OR}}$ are not perfect it is possible to deduce some properties of $D^{\mathcal{OR}}$ by analyzing its semantic counterpart D_{Tr}. Let D_{Tr} denote the semantic malware detection algorithm which is equivalent to the detection scheme $D^{\mathcal{OR}}$ working on perfect oracles. In general, by relaxing the hypothesis of perfect oracles, we have that the malware detector $D^{\mathcal{OR}}$ is less precise than its (ideal) semantic counterpart D_{Tr}. This means that:

- $D^{\mathcal{OR}}$ is sound with respect to obfuscation $\mathcal{O} \Rightarrow D_{Tr}$ is sound with respect to obfuscation $\mathcal{O}$
- $D^{\mathcal{OR}}$ is complete with respect to obfuscation $\mathcal{O} \Rightarrow D_{Tr}$ is complete with respect to obfuscation $\mathcal{O}$

In this case, by proving that D_{Tr} is not sound (resp. complete) with respect to a given obfuscation $\mathcal{O}$ we can deduce that $D^{\mathcal{OR}}$ is not sound (resp. complete) with respect to $\mathcal{O}$ as well. On the other hand, even if we are able to prove that D_{Tr} is sound or complete with respect to an obfuscation $\mathcal{O}$ we cannot infer anything about the soundness or completeness of $D^{\mathcal{OR}}$ with respect to $\mathcal{O}$.

Under the assumption of perfect oracles, in Section 6.7 we apply the proof strategy presented above to the semantics-aware malware detector proposed by Christodorescu et al. [25], and in Section 6.6 to the standard signature matching approach.

6.1.2 Programming Language

The language considered in this chapter is a simple extension of the one introduced by Cousot and Cousot [44] (and described in Section 2.3), the main difference being the ability of programs to generate code dynamically (this facility is added to accommodate certain kinds of malware obfuscations where the payload is unpacked and decrypted at runtime). The syntax of our language is given in Table 6.1. As usual, given a set S, we use $S_\perp$ to denote the set $S \cup \{\perp\}$, where $\perp$ denotes an undefined value. Assume that program variables can store either an integer value or a command, encoded as a pair (A, S), where A and S correspond respectively to the action and the successor labels of the stored command. This leads to the introduction of the syntactic category $\mathbb{E} \cup (\mathbb{A} \times \wp(\mathbb{L}))$ representing the set of possible assignment r-values. Commands can be either

Syntactic Categories:		**Syntax:**	
$n \in \mathbb{Z}$	(integers)	$E ::=$	$n \mid X \mid E_1$ `op` E_2
$X \in \mathbb{X}$	(variable names)		(`op` $\in \{+, -, *, /, \ldots\}$)
$L \in \mathbb{L}$	(labels)	$B ::=$	`true` $\mid$ `false` $\mid E_1 < E_2$
$E \in \mathbb{E}$	(integer expr.)		$\mid \neg B_1 \mid B_1$ && B_2
$B \in \mathbb{B}$	(Boolean expr.)	$A ::=$	$X := D \mid$ `skip` $\mid$ `assign`(L, X)
$A \in \mathbb{A}$	(actions)	$C ::=$	$L : A \to L'$ (unconditional)
$D \in \mathbb{E} \cup (\mathbb{A} \times \wp(\mathbb{L}))$	(assignment r-values)		$L : B \to \{L_T, L_F\}$ (conditional)
$C \in \mathbb{C}$	(commands)	$P ::=$	$\wp(\mathbb{C})$
$P \in \mathbb{P}$	(programs)		

Table 6.1. Syntax of the programming language

conditional or unconditional. A conditional command at a label L has the form $L : B \to \{L_T, L_F\}$, where B is a Boolean expression and L_T (respectively, L_F) is the label of the command to execute when B evaluates to *true* (respectively, *false*); an unconditional command at a label L is of the form $L : A \to L_1$, where A is an action and L_1 the label of the command to be executed next. As observed earlier, a variable can be undefined ($\perp$), or it can store either an integer or an (appropriately encoded) pair $(A, S) \in \mathbb{A} \times \wp(\mathbb{L})$. The auxiliary functions in Table 6.2 are useful in defining the semantics of the considered programming language, which is described in Table 6.3. A program consists of an initial set of

Labels	**Successors of a command**
$lab[\![L : A \to L']\!] \stackrel{\text{def}}{=} L$ $lab[\![L : B \to \{L_T, L_F\}]\!] \stackrel{\text{def}}{=} L$ $lab[\![P]\!] \stackrel{\text{def}}{=} \{lab[\![C]\!] \mid C \in P\}$	$suc[\![L : A \to L']\!] \stackrel{\text{def}}{=} L'$ $suc[\![L : B \to \{L_T, L_F\}]\!] \stackrel{\text{def}}{=} \{L_T, L_F\}$
Variables	**Memory locations used by a program**
$var[\![L_1 : A \to L_2]\!] \stackrel{\text{def}}{=} var[\![A]\!]$ $var[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{C \in P} var[\![C]\!]$ $var[\![A]\!] = \{\text{variables occurring in } A\}$	$Luse[\![L : A \to L']\!] \stackrel{\text{def}}{=} Luse[\![A]\!]$ $Luse[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{C \in P} Luse[\![C]\!]$ $Luse[\![A]\!] = \{\text{locations occurring in } A\} \cup \rho(var[\![A]\!])$
Action of a command	**Commands in sequences of program states**
$act[\![L : A \to L_2]\!] \stackrel{\text{def}}{=} A$	$cmd[\![\{(C_1, \xi_1), \ldots, (C_k, \xi_k)\}]\!] = \{C_1, \ldots, C_k\}$

Table 6.2. Auxiliary functions

commands together with all the commands that are reachable through execution from the initial set. In other words, if P_{init} denotes the initial set of commands, then $P = cmd[\![\bigcup_{C \in P_{init}} \left(\bigcup_{\xi \in \mathfrak{X}} \mathbf{C}^*(C, \xi)\right)]\!]$, where we extend the transition relation $\mathbf{C}$ to a set of program states, i.e., $\mathbf{C}(S) = \bigcup_{\sigma \in S} \mathbf{C}(\sigma)$. Since each command

Value Domains	
$\mathcal{B} = \{true, false\}$	(truth values)
$n \in \mathbb{Z}$	(integers)
$\rho \in \mathfrak{E} = \mathbb{X} \to \mathbb{L}_\perp$	(environments)
$m \in \mathfrak{M} = \mathbb{L} \to \mathbb{Z}_\perp \cup (\mathbb{A} \times \wp(\mathbb{L}))$	(memories)
$\xi \in \mathfrak{X} = \mathfrak{E} \times \mathfrak{M}$	(execution contexts)
$\Sigma = \mathbb{C} \times \mathfrak{X}$	(program states)
Arithmetic Expressions	$\mathbf{E} : \mathbb{A} \times \mathfrak{X} \to \mathbb{Z}_\perp \cup (\mathbb{A} \times \wp(\mathbb{L}))$
$\mathbf{E}[\![n]\!]\xi$	$= n$
$\mathbf{E}[\![X]\!]\xi$	$= m(\rho(X))$, where $\xi = (\rho, m)$
$\mathbf{E}[\![E_1 \texttt{ op } E_2]\!]\xi$	$= \begin{cases} \mathbf{E}[\![E_1]\!]\xi \texttt{ op } \mathbf{E}[\![E_2]\!]\xi & \text{if } \mathbf{E}[\![E_1]\!]\xi, \mathbf{E}[\![E_2]\!]\xi \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$
Boolean expressions	$\mathbf{B} : \mathbb{B} \times \mathfrak{X} \to \mathcal{B}_\perp$
$\mathbf{B}[\![\texttt{true}]\!]\xi$	$= true$
$\mathbf{B}[\![\texttt{false}]\!]\xi$	$= false$
$\mathbf{B}[\![E_1 < E_2]\!]\xi$	$= \begin{cases} \mathbf{E}[\![E_1]\!]\xi < \mathbf{E}[\![E_2]\!]\xi & \text{if } \mathbf{E}[\![E_1]\!]\xi, \mathbf{E}[\![E_2]\!]\xi \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$
$\mathbf{B}[\![\neg B]\!]\xi$	$=$ if $(\mathbf{B}[\![B]\!]\xi \in \mathcal{B})$ then $\neg\mathbf{B}[\![B]\!]\xi$; else $\perp$
$\mathbf{B}[\![B_1 \texttt{ \&\& } B_2]\!]\xi$	$= \begin{cases} \mathbf{B}[\![B_1]\!]\xi \wedge \mathbf{B}[\![B_2]\!]\xi & \text{if } \mathbf{B}[\![B_1]\!]\xi, [\![B_2]\!]\xi \in \mathcal{B} \\ \perp & \text{otherwise} \end{cases}$
Actions	$\mathbf{A} : \mathbb{A} \times \mathfrak{X} \to \mathfrak{X}$
$\mathbf{A}[\![\texttt{skip}]\!]\xi$	$= \xi$
$\mathbf{A}[\![X := D]\!]\xi$	$= (\rho, m')$, where $\xi = (\rho, m), m' = m[\rho(X) \leftarrow \delta]$ and $\delta = \begin{cases} D & \text{if } D \in \mathbb{A} \times \wp(\mathbb{L}) \\ \mathbf{E}[\![D]\!](\rho, m) & \text{if } D \in \mathbb{E} \end{cases}$
$\mathbf{A}[\![\texttt{assign}(L', X)]\!]\xi$	$= (\rho', m)$, where $\xi = (\rho, m)$ and $\rho' = \rho[X \rightsquigarrow L']$
Commands	$\mathbf{C} : \Sigma \to \wp(\Sigma)$
$\mathbf{C}[\![L : A \to L']\!]\xi$	$= \{(C, \xi') \mid \xi' = \mathbf{A}[\![A]\!]\xi, lab[\![C]\!] = L', \langle act[\![C]\!] : suc[\![C]\!]\rangle = m'(L')\}$, where $\xi' = (\rho', m')$
$\mathbf{C}[\![L : B \to \{L_T, L_F\}]\!]\xi$	$= \{(C, \xi) \mid lab[\![C]\!] = \begin{cases} L_T \text{ if } \mathbf{B}[\![B]\!]\xi = true \\ L_F \text{ if } \mathbf{B}[\![B]\!]\xi = false \end{cases} \wedge \langle act[\![C]\!] : suc[\![C]\!]\rangle = \begin{cases} m(L_T) \text{ if } \mathbf{B}[\![B]\!]\xi = true \\ m(L_F) \text{ if } \mathbf{B}[\![B]\!]\xi = false \end{cases}\}$

Table 6.3. Semantics of the programming language

explicitly mentions its successors, the program need not to maintain an explicit sequence of commands. This definition allows us to represent programs that generate code dynamically.

An *environment* $\rho \in \mathfrak{E}$ maps variables in $dom(\rho) \subseteq \mathbb{X}$ to memory locations $\mathbb{L}_\perp$. Given a program P we denote with $\mathfrak{E}(P)$ its environments, i.e., if $\rho \in \mathfrak{E}(P)$ then $dom(\rho) = var[\![P]\!]$. Let $\rho[X \rightsquigarrow L]$ denote environment ρ where label L is assigned to variable X. The *memory* is represented as a function $m : \mathbb{L} \to \mathbb{Z}_\perp \cup (\mathbb{A} \times \wp(\mathbb{L}))$. Let $m[L \leftarrow D]$ denote memory m where element D is stored at location L. When considering a program P, we denote with $\mathfrak{M}(P)$ the set of program memories, namely if $m \in \mathfrak{M}(P)$ then $dom(m) = Luse[\![P]\!]$. This means that $m \in \mathfrak{M}(P)$ is defined on the set of memory locations that are affected by the execution of program P (excluding the memory locations storing the initial commands of P).

The behavior of a command when it is executed depends on its *execution context*, i.e., the environment and memory in which it is executed. The set of execution contexts is given by $\mathfrak{X} = \mathfrak{E} \times \mathfrak{M}$. A *program state* is a pair (C, ξ) where C is the next command that has to be executed in the execution context ξ. $\Sigma = \mathbb{C} \times \mathfrak{X}$ denotes the set of all possible states. Given a state $s \in \Sigma$, the semantic function $\mathbf{C}(s)$ gives the set of possible successor states of s; in other words, $\mathbf{C} : \Sigma \to \wp(\Sigma)$ defines the transition relation between states. Let $\Sigma(P) = P \times \mathfrak{X}(P)$ be the set of states of a program P, then we can specify the transition relation on program P as $\mathbf{C}[\![P]\!] : \Sigma(P) \to \wp(\Sigma(P))$:

$$\mathbf{C}[\![P]\!](C, \xi) \stackrel{\text{def}}{=} \left\{ \; (C', \xi') \;\middle|\; (C', \xi') \in \mathbf{C}(C, \xi), C' \in P, \text{ and } \xi, \xi' \in \mathfrak{X}(P) \; \right\}$$

Let A^* denote the Kleene closure of a set A, i.e., the set of finite sequences over A. A *trace* $\sigma \in \Sigma^*$ is a sequence of states $s_1 ... s_n$ of length $|\sigma| \geq 0$ such that for all $i \in [1, n)$: $s_i \in \mathbf{C}(s_{i-1})$. The *finite partial traces semantics* $\mathbf{S}[\![P]\!] \subseteq \Sigma^*$ of program P is the least fixpoint of the function F:

$$F[\![P]\!](T) \stackrel{\text{def}}{=} \Sigma(P) \cup \{ss'\sigma | s' \in \mathbf{C}[\![P]\!](s), \; s'\sigma \in T\}$$

where T is a set of traces, namely $\mathbf{S}[\![P]\!] = lfp^{\subseteq} F[\![P]\!]$. The set of all partial trace semantics, ordered by set inclusion, forms a complete lattice.

6.2 Semantics-Based Malware Detection

In this section we introduce a formalization of the malware detection problem based on program semantics and abstract interpretation. Intuitively, a program P is infected by a malware M if (part of) P's execution behavior is similar to that of M, namely if there is a moment during the execution of program P where malware M is executed. Therefore, in order to detect the presence of a malicious

behavior from a malware M in a program P, we need to check whether there is a part (i.e., a restriction) of program semantics $\mathbf{S}[\![P]\!]$ that "matches" (in a sense that will be made precise) the malware semantics $\mathbf{S}[\![M]\!]$. In the following we show how program restriction as well as semantic matching can actually be expressed as abstractions of program semantics in the abstract interpretation sense.

It is clear how the process of considering only a portion of program semantics can be seen as an abstraction of $\mathbf{S}[\![P]\!]$. A subset of a program P's labels (i.e., commands) $lab_r[\![P]\!] \subseteq lab[\![P]\!]$ characterizes a *restriction* of program P. In particular, let $var_r[\![P]\!]$ and $Luse_r[\![P]\!]$ denote, respectively, the set of variables occurring in the restriction and the set of memory locations used in the restriction:

$$var_r[\![P]\!] \stackrel{\text{def}}{=} \bigcup \{\ var[\![C]\!] \mid lab[\![C]\!] \in lab_r[\![P]\!]\ \}$$
$$Luse_r[\![P]\!] \stackrel{\text{def}}{=} \bigcup \{\ Luse[\![C]\!] \mid lab[\![C]\!] \in lab_r[\![P]\!]\ \}$$

Thus, the set of labels $lab_r[\![P]\!]$ induces a restriction on environment and memory maps. Given $\rho \in \mathfrak{E}(P)$ and $m \in \mathfrak{M}(P)$, let $\rho^r \stackrel{\text{def}}{=} \rho_{|var_r[\![P]\!]}$ and $m^r \stackrel{\text{def}}{=} m_{|Luse_r[\![P]\!]}$ denote the restricted set of environments and memories induced by the restricted set of labels $lab_r[\![P]\!]$. Let $\Sigma_r = \{(C, \rho^r, m^r) \mid lab[\![C]\!] \in lab_r[\![P]\!]\}$ be the set of restricted program states. Let us define abstraction $\alpha_r : \Sigma^* \to \Sigma_r^*$ that propagates restriction $lab_r[\![P]\!]$ on a given a trace $\sigma = (C_1, \rho_1, m_1)\sigma'$:

$$\alpha_r(\sigma) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ (C_1, \rho_1^r, m_1^r)\alpha_r(\sigma') & \text{if } lab[\![C_1]\!] \in lab_r[\![P]\!] \\ \alpha_r(\sigma') & \text{otherwise} \end{cases}$$

Given a function $f : A \to B$ we denote, by a slight abuse of notation, its pointwise extension on powerset as $f : \wp(A) \to \wp(B)$, where $f(X) \stackrel{\text{def}}{=} \{f(x) | x \in X\}$. Note that the pointwise extension is additive. Therefore, the function $\alpha_r : \wp(\Sigma^*) \to \wp(\Sigma_r^*)$ can be seen as an abstraction that discards information outside the restriction $lab_r[\![P]\!]$. Moreover, α_r is surjective and defines a Galois insertion:

$$\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_r}{\overset{\gamma_r}{\leftrightarrows}} \langle \wp(\Sigma_r^*), \subseteq \rangle$$

Let $\alpha_r(\mathbf{S}[\![P]\!])$ be the *restricted semantics* of program P. Given a program P and a restriction $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$, let $P_r \stackrel{\text{def}}{=} \{C \in P | lab[\![C]\!] \in lab_r[\![P]\!]\}$ be the program obtained by considering only the commands of P with labels in $lab_r[\![P]\!]$. If P_r is a program, namely if it is possible to compute its semantics, then $\mathbf{S}[\![P_r]\!](I) = \alpha_r(\mathbf{S}[\![P]\!])$, where I is the set of possible states of program P when P executes the first command in P_r.

Let us observe that the effects of program execution on the execution context, i.e., on environments and memories, express program behaviour more than the

particular commands that cause such effects (in fact different sequences of commands may produce the same sequence of modifications on environments and memories). For this reason, let us consider the transformation $\alpha_e : \Sigma^* \to \mathfrak{X}^*$ that, given a trace σ, discards from σ all information about the commands that are executed, retaining only the information about the changes in the execution context (i.e., in environments and memories).

$$\alpha_e(\sigma) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \xi_1 \alpha_e(\sigma') & \text{if } \sigma = (C_1, \xi_1)\sigma' \end{cases}$$

Two traces σ and δ in Σ^* are considered "similar" if they are the same under α_e, namely if they have the same sequence of effects on environments and memories, i.e., if $\alpha_e(\sigma) = \alpha_e(\delta)$. This *semantic matching* relation between program traces is the basis of our approach to malware detection. The additive function $\alpha_e : \wp(\Sigma^*) \to \wp(\mathfrak{X}^*)$ abstracts from the trace semantics of a program and defines a Galois insertion:

$$\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} \langle \wp(\mathfrak{X}^*), \subseteq \rangle$$

Let us say that a malware is a *vanilla malware* if no obfuscating transformations have been applied to it. The following definition provides a semantic characterization of the presence of a vanilla malware M in a program P in terms of the semantic abstractions α_r and α_e.

Definition 6.4. A program P is *infected* by a vanilla malware M, i.e., $M \hookrightarrow P$, if:

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_e(\mathbf{S}[\![M]\!]) \subseteq \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$$

A *semantic malware detector* is a system that verifies the presence of a malware in a program by checking the truth of the inclusion relation of the above definition. Following this definition, a program P is classified as infected by a vanilla malware M, if P exhibits behaviors that, under abstractions α_r and α_e, match all of the behaviors of M. It is clear that this is a strong requirement and that the notion of semantic infection can be weakened. In fact, in Section 6.5, we will consider a weaker notion of malware infection, where only some (not all) behaviors of the malware are present in the program.

6.3 Obfuscated Malware

We have argued above how malware writers usually obfuscate the malicious code in order to prevent detection. Thus, a robust malware detector needs to handle possibly obfuscated versions of a malware. While obfuscation may modify the original code, the obfuscated code has to be equivalent (up to some notion of equivalence) to the original one. Given an obfuscating transformation $\mathcal{O} : \mathbb{P} \to \mathbb{P}$

on programs our idea is to design a suitable abstract domain A, such that the abstraction $\alpha : \wp(\mathfrak{X}^*) \to A$ discards the details changed by the obfuscation while preserving the maliciousness of the program. The main idea is that, different obfuscated versions of a program are equivalent up to $\alpha \circ \alpha_e$. Hence, in order to verify program infection, we check whether there exists a semantic program restriction that matches the malware behavior up to α, formally:

$$\exists\, lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))) \tag{6.1}$$

Here $\alpha_r(\mathbf{S}[\![P]\!])$ is the restricted semantics of program P; $\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$ retains only the environment-memory traces from the restricted semantics; and α further discards any effects due to obfuscation. We then check that the resulting set of environment-memory traces contains all of the environment-memory traces from the malware semantics, with obfuscation effects abstracted away via α. In this setting, abstraction α allows us to ignore obfuscation and concentrate on the malicious intent. A semantic malware detector on α is a detection algorithm that verifies program infection according to 6.1.

Example 6.5. Let us consider the fragment of program P that computes the factorial of variable X and its obfuscation $\mathcal{O}[\![P]\!]$ obtained by inserting commands that do not affect the execution context (at labels L_2 and L_{F+1} in the example).

P

L_1	$: F := 1 \to L_2$
L_2	$: (X = 1) \to \{L_T, L_F\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: F := F \times X \to L_2$
L_T	$: ...$

$\mathcal{O}[\![P]\!]$

L_1	$: F := 1 \to L_2$
L_2	$: F := F \times 2 - F \to L_3$
L_3	$: (X = 1) \to \{L_T, L_F\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: X := X \times 1 \to L_{F+2}$
L_{F+2}	$: F := F \times X \to L_3$
L_T	$: ....$

It is clear that $\mathbf{A}[\![F := F \times 2 - F]\!]\xi = \xi$ and $\mathbf{A}[\![X := X \times 1]\!]\xi = \xi$ for all $\xi \in \mathfrak{X}$. Thus, a suitable abstraction α in order to deal with the insertion of such semantic NOP commands, is the one that observes modifications in the execution context, formally let $\xi_i = (\rho_i, m_i)$:

$$\alpha(\xi_1, \xi_2, ..., \xi_n) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } \xi_1, \xi_2, ..., \xi_n = \epsilon \\ \alpha(\xi_2, ..., \xi_n) & \text{if } \xi_1 = \xi_2 \\ \xi_1 \alpha(\xi_2, ..., \xi_n) & \text{otherwise} \end{cases}$$

In fact it is possible to show that $\alpha(\alpha_e(\mathbf{S}[\![P]\!])) = \alpha(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![P]\!]]\!]))$.

□

The extent to which a semantic malware detector is able to discriminate between infected and uninfected code, and therefore the balance between any false positives and any false negatives it may incur, depends on the abstraction function α. On one side, augmenting the degree of abstraction of α increases the ability of the detector to deal with obfuscation but, at the same time, increases the false positive rate, namely the number of programs erroneously classified as infected. On the other side, a more concrete α makes the detector more sensitive to obfuscation, while decreasing the presence of programs miss-classified as infected. In the following we provide a semantic characterization of the notions of soundness and completeness, introduced in Definition 6.1.

Definition 6.6.

- A semantic malware detector on α is *complete* for a set $\mathbb{O}$ of transformations if and only if $\forall \mathcal{O} \in \mathbb{O}$:

$$\mathcal{O}[\![M]\!] \hookrightarrow P \Rightarrow \begin{cases} \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \\ \alpha(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))) \end{cases}$$

- A semantic malware detector on α is *sound* for a set $\mathbb{O}$ of transformations if and only if:

$$\left.\begin{array}{l} \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \\ \alpha(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))) \end{array}\right\} \Rightarrow \exists \mathcal{O} \in \mathbb{O} : \mathcal{O}[\![M]\!] \hookrightarrow P$$

In particular, completeness for a class $\mathbb{O}$ of obfuscating transformations means that, for every obfuscation $\mathcal{O} \in \mathbb{O}$, when program P is infected by a variant $\mathcal{O}[\![M]\!]$ of a malware, then the semantic malware detector is able to detect it (i.e., no false negatives). On the other side, soundness with respect to the class $\mathbb{O}$ of obfuscating transformations, means that when the semantic malware detector classifies a program P as infected by a malware M, then there exists an obfuscation $\mathcal{O} \in \mathbb{O}$, such that program P is infected by the variant $\mathcal{O}[\![M]\!]$ of the malware (i.e., no false positives). In the following, when considering a class $\mathbb{O}$ of obfuscating transformations, we will assume that also the identity function belongs to $\mathbb{O}$, in this way we include in the set of variants identified by $\mathbb{O}$ the malware itself. It is interesting to observe that, considering an obfuscating transformation $\mathcal{O}$, completeness is guaranteed when abstraction α is preserved by $\mathcal{O}$, namely when $\forall P \in \mathbb{P} : \alpha(\alpha_e(\mathbf{S}[\![P]\!])) = \alpha(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![P]\!]]\!]))$.

Theorem 6.7. If abstraction $\alpha : \wp(\mathfrak{X}^*) \to A$ is preserved by transformation $\mathcal{O}$, then the semantic malware detector on α is complete for $\mathcal{O}$.

PROOF: In order to show that the semantic malware detector on α is complete for $\mathcal{O}$, we have to show that if $\mathcal{O}[\![M]\!] \hookrightarrow P$ then there exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. If $\mathcal{O}[\![M]\!] \hookrightarrow P$, it means that there

exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $P_r = \mathcal{O}[\![M]\!]$. By definition $\mathcal{O}[\![M]\!]$ is a program and therefore $\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!] = \mathbf{S}[\![P_r]\!] = \alpha_r(\mathbf{S}[\![P]\!])$. Moreover, we have that $\alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))) = \alpha(\alpha_e(\mathbf{S}[\![P_r]\!])) = \alpha(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!])) = \alpha(\alpha_e(\mathbf{S}[\![M]\!]))$, where the last equality follows from the hypothesis that α is preserved by $\mathcal{O}$. Thus, $\alpha(\alpha_e(\mathbf{S}[\![M]\!])) = \alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$ which concludes the proof.

□

However, the preservation condition of Theorem 6.7 is too weak to imply soundness of the semantic malware detector. As an example let us consider the abstraction $\alpha_\top = \lambda X.\top$ that loses all information. It is clear that $\alpha_\top$ is preserved by every obfuscating transformation, but the semantic malware detector on $\alpha_\top$ classifies every program as infected by every malware. Unfortunately, we do not have a result analogous to Theorem 6.7 that provides a property of abstraction α that characterizes soundness of the semantic malware detector. However, given an abstraction α, we can characterize the set of transformations for which α is sound.

Theorem 6.8. Given an abstraction α, consider the set $\mathbb{O}$ of transformations such that: $\forall P, Q \in \mathbb{P}$:

$$(\alpha(\alpha_e(\mathbf{S}[\![Q]\!])) \subseteq \alpha(\alpha_e(\mathbf{S}[\![P]\!]))) \;\Rightarrow\; (\exists \mathcal{O} \in \mathbb{O} : \alpha_e(\mathbf{S}[\![\mathcal{O}[\![Q]\!]]\!]) \subseteq \alpha_e(\mathbf{S}[\![P]\!]))$$

Then, a semantic malware detector on α is sound for $\mathbb{O}$.

PROOF: Suppose that these exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$, since $M, P, P_r \in \mathbb{P}$ and $\alpha_r(\mathbf{S}[\![P]\!]) = \mathbf{S}[\![P_r]\!]$, then by definition of set $\mathbb{O}$ we have that: $\exists \mathcal{O} \in \mathbb{O} : \alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]) \subseteq \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$, and therefore $\mathcal{O}[\![M]\!] \hookrightarrow P$.

□

6.4 A Semantic Classification of Obfuscations

In this section we classify obfuscating transformations according to their effects on program trace semantics. In particular, we distinguish between transformations that add new instructions while maintaining the structure of the original program traces, and transformations that insert new instructions causing major changes to the original semantic structure. Given two sequences $s, t \in A^*$ for some set A, let $s \preceq t$ denote that s is a subsequence of t, i.e., if $s = s_1 s_2 \dots s_n$ then t is of the form $\dots s_1 \dots s_2 \dots s_n \dots$.

6.4.1 Conservative Obfuscations

An obfuscating transformation $\mathcal{O} : \mathbb{P} \to \mathbb{P}$ is a *conservative obfuscation* if every trace σ of the original program semantics is a subsequence of some trace δ of the obfuscated program semantics, formally, if:

$$\forall \sigma \in \mathbf{S}[\![P]\!], \exists \delta \in \mathbf{S}[\![\mathcal{O}[\![P]\!]]\!] : \alpha_e(\sigma) \preceq \alpha_e(\delta)$$

Let $\mathbb{O}_c$ denote the set of conservative obfuscating transformations. When dealing with conservative obfuscations, we have that a trace δ of a program P presents a possibly obfuscated malicious behavior M, if there is a malware trace $\sigma \in \mathbf{S}[\![M]\!]$ whose environment-memory evolution is "contained" in the environment-memory evolution of δ, namely if $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Let us define the abstraction $\alpha_c : \wp(\mathfrak{X}^*) \to (\mathfrak{X}^* \to \wp(\mathfrak{X}^*))$ that, given a context sequence $s \in \mathfrak{X}^*$ and a set of context sequences $S \in \wp(\mathfrak{X}^*)$, returns the elements $t \in S$ that are subsequences of s:

$$\alpha_c[S](s) \stackrel{\text{def}}{=} S \cap \mathit{SubSeq}(s)$$

where $\mathit{SubSeq}(s) \stackrel{\text{def}}{=} \{t | t \preceq s\}$ denotes the set of all subsequences of s. For any $S \in \wp(\mathfrak{X}^*)$, the additive function $\alpha_c[S]$ defines a Galois connection:

$$\langle \wp(\mathfrak{X}^*), \subseteq \rangle \underset{\alpha_c[S]}{\overset{\gamma_c[S]}{\leftrightarrows}} \langle \wp(\mathfrak{X}^*), \subseteq \rangle$$

The abstraction α_c turns out to be a suitable approximation when dealing with conservative obfuscations. In fact, the semantic malware detector on $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])]$ is complete and sound with respect to the class of conservative obfuscations $\mathbb{O}_c$.

Theorem 6.9. Considering a vanilla malware M we have that a semantic malware detector on $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])]$ is complete and sound for $\mathbb{O}_c$, namely:
Completeness:

$$\left.\begin{array}{l} \forall \mathcal{O}_c \in \mathbb{O}_c : \\ \mathcal{O}_c[\![M]\!] \hookrightarrow P \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \exists \mathit{lab}_r[\![P]\!] \in \wp(\mathit{lab}[\![P]\!]) : \\ \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))) \end{array}\right.$$

Soundness:

$$\left.\begin{array}{l} \exists \mathit{lab}_r[\![P]\!] \in \wp(\mathit{lab}[\![P]\!]) : \\ \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \exists \mathcal{O}_c \in \mathbb{O}_c : \\ \mathcal{O}_c[\![M]\!] \hookrightarrow P \end{array}\right.$$

PROOF: Completeness: Let $\mathcal{O}_c \in \mathbb{O}_c$, if $\mathcal{O}_c[\![M]\!] \hookrightarrow P$ it means that $\exists\ \mathit{lab}_r[\![P]\!] \in \wp(\mathit{lab}[\![P]\!])$ such that $P_r = \mathcal{O}_c[\![M]\!]$. Such restriction is the one that satisfies the condition on the right. In fact, $P_r = \mathcal{O}_c[\![M]\!]$ means that $\alpha_r(\mathbf{S}[\![P]\!]) = \mathbf{S}[\![\mathcal{O}_c[\![M]\!]]\!]$. We have to show: $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![\mathcal{O}_c[\![M]\!]]\!]))$. By definition of conservative obfuscation for each trace $\sigma \in \mathbf{S}[\![M]\!]$ there exists

$\delta \in \mathbf{S}[\![\mathcal{O}_c[\![M]\!]]\!]$ such that: $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Considering such σ and δ we show that $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\delta))$, in fact:

$$\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\delta)) = \alpha_e(\mathbf{S}[\![M]\!]) \cap \mathit{SubSeq}(\alpha_e(\delta))$$
$$\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\sigma)) = \alpha_e(\mathbf{S}[\![M]\!]) \cap \mathit{SubSeq}(\alpha_e(\sigma))$$

Since $\alpha_e(\sigma) \preceq \alpha_e(\delta)$, it follows that $\mathit{SubSeq}(\alpha_e(\sigma)) \subseteq \mathit{SubSeq}(\alpha_e(\delta))$. Therefore, $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\delta))$, which concludes the proof.
Soundness: By hypothesis there exists $\mathit{lab}_r[\![P]\!] \in \wp(\mathit{lab}[\![P]\!])$ for which it holds that $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. This means that $\forall \sigma \in \mathbf{S}[\![M]\!]$ we have that: $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$, which means that $\alpha_e(\sigma) \in \{\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\delta)) \mid \delta \in \alpha_r(\mathbf{S}[\![P]\!])\}$. Thus, $\forall \sigma \in \mathbf{S}[\![M]\!]$, there exists $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and this means that P_r is a conservative obfuscation of malware M, namely $\exists \mathcal{O}_c \in \mathbb{O}_c$ such that $\mathcal{O}_c[\![M]\!] \hookrightarrow P$.

□

It turns out that many obfuscating transformations commonly used by malware writers are conservative; a partial list of such conservative obfuscations is given below. For each transformation we provide a simple example and a sketch proof of their conservativeness. It follows that Theorem 6.9 is applicable to a significant class of malware-obfuscation transformations.

Code reordering

This transformation, commonly used to avoid signature matching detection, changes the order in which commands are written, while maintaining the execution order through the insertion of unconditional jumps (see Fig. 6.1 for an example).

P

L_1	$: F := 1 \to L_2$
L_2	$: (X = 1) \to \{L_T, L_F\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: F := F \times X \to L_2$
L_T	$: ...$

$\mathcal{O}_J[\![P]\!]$

L_1	$: F := 1 \to L_2$
L_2	$:$ `skip` $\to L_3$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$:$ `skip` $\to L_4$
L_3	$: (X = 1) \to \{L_T, L_F\}$
L_T	$: ....$
L_4	$: F := F \times 2 - F \to L_3$

Fig. 6.1. Code reordering

Observe that, in the programming language introduced in Section 6.1.2, an unconditional jump is expressed as a command $L :$ `skip` $\to L'$ that directs the

flow of control of the program to a command labelled by L'. Let P be a program, $P = \{C_i : 1 \leq i \leq N\}$. The code reordering obfuscating transformation $\mathcal{O}_J : \mathbb{P} \to \mathbb{P}$ inserts $L : \texttt{skip} \to L'$ commands after selected commands from the program P. Let $R \subseteq P$ be a set of $m \leq N$ commands selected by the obfuscating transformation $\mathcal{O}_J$, i.e., $|R| = m$. The `skip` commands are then inserted after each one of the m selected commands in R. Let us define the subset S of commands of P that contains the successors of the commands in R:

$$S \stackrel{\text{def}}{=} \left\{ C' \in P \,\middle|\, \exists C \in R \ : lab[\![C']\!] \in suc[\![C]\!] \right\}$$

Effectively, the code reordering obfuscating transformation adds a `skip` between a command $C \in R$ and its successor $C' \in S$. Define $\eta : \mathbb{C} \to \mathbb{C}$, a command-relabeling function, as follows:

$$\eta\,(L_1 : A \to L_2) \stackrel{\text{def}}{=} \mathit{NewLabel}(\mathbb{L} \setminus \{L_1\}) : A \to L_2$$

where $\mathit{NewLabel}(H)$ returns a label from the set $H \subseteq \mathbb{L}$. We extend η to a set of commands $T = \{\ldots, L_i : A \to L_j, \ldots\}$:

$$\eta(T) \stackrel{\text{def}}{=} \left\{ \ldots, \mathit{NewLabel}(\mathbb{L}') : A \to L_j, \ldots \right\}$$

where $\mathbb{L}' = \mathbb{L} \setminus \{\ldots, L_i, \ldots\}$. We can define the set of `skip` commands inserted by this obfuscating transformation:

$$\mathit{Skip}(S) \stackrel{\text{def}}{=} \left\{ L : \texttt{skip} \to L' \,\middle|\, \exists C \in S : L = lab[\![C]\!], L' = lab[\![\eta(C)]\!] \right\}$$

Then, $\mathcal{O}_J[\![P]\!] = (P \setminus S) \cup \eta(S) \ \cup \mathit{Skip}(S)$. Observing the effects that code reordering has on program semantics we have that for each trace $\sigma \in \mathbf{S}[\![P]\!]$, where $\sigma = \langle C_1, \rho_1, m_1\rangle ... \langle C_n, \rho_n, m_n\rangle$, there exists an obfuscated trace $\delta \in \mathbf{S}[\![\mathcal{O}_J[\![P]\!]]\!]$ such that $\delta = \langle SK, \rho_1, m_1\rangle^* \langle C'_1, \rho_1, m_1\rangle \ldots \langle SK, \rho_n, m_n\rangle^* \langle C'_n, \rho_n, m_n\rangle$, where $act[\![C_i]\!] = act[\![C'_i]\!]$ and $SK \in \mathit{Skip}(S)$. Thus, $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and $\mathcal{O}_J \in \mathbb{O}_c$.

Opaque predicate insertion

This program transformation confuses the original control flow of the program by inserting opaque predicates, i.e., a predicate whose value is known a priori to a program transformation but is difficult to determine by examining the transformed program [31]. In the following, we given an idea of way opaque predicate insertion is a conservative transformation, considering the three major types of opaque predicates: true, false and unknown (see Fig. 6.2 for an example of true opaque predicate insertion). In the considered programming language a *true opaque predicate* is expressed by a command $L : P^T \to \{L_T, L_F\}$. Since P^T always evaluate *true* the next command label is always L_T. When a true opaque

P	
L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

$\mathcal{O}_T[\![P]\!]$	
L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_O\}$
L_O	$: P^T \rightarrow \{L_F, L_B\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_B	: buggy code
L_T	$: ...$

Fig. 6.2. True opaque predicate insertion at label L_O

predicate is inserted after command C the sequence of commands starting at label L_T is the sequence starting at $suc[\![C]\!]$ in the original program, while some buggy code is inserted starting form label L_F. Let $\mathcal{O}_T : \mathbb{P} \rightarrow \mathbb{P}$ be the obfuscating transformation that inserts true opaque predicates, and let P, R, S and η be defined as in the code reordering case. In fact, transformation $\mathcal{O}_T$ inserts opaque predicates between a command C in R and its successor C' in S. Let us define the set of commands encoding opaque predicate P^T inserted by $\mathcal{O}_T$ as:

$$TrueOp(S) \stackrel{\text{def}}{=} \left\{ L : P^T \rightarrow \{L_T, L_F\} \;\middle|\; \begin{array}{l} \exists C \in S : \\ L = lab[\![C]\!], L_T = lab[\![\eta(C)]\!] \end{array} \right\}$$

$$Bug(TrueOp(S)) \stackrel{\text{def}}{=} \left\{ B_1...B_k \;\middle|\; \begin{array}{l} B_1...B_k \in \wp(\mathbb{C}) \\ \exists L : P^T \rightarrow \{L_T, L_F\} \in TrueOp(S) : \\ lab[\![B_1]\!] = L_F \end{array} \right\}$$

where $B_1...B_k$ is a sequence of commands expressing some buggy code. Then:

$$\mathcal{O}_T[\![P]\!] = (P \setminus S) \cup \eta(S) \cup TrueOp(S) \cup Bug(TrueOp(S))$$

Observing the effects on program semantics we have that for each trace $\sigma \in \mathbf{S}[\![P]\!]$, such that $\sigma = \langle C_1, \rho_1, m_1\rangle ... \langle C_n, \rho_n, m_n\rangle$ there exists $\delta \in \mathbf{S}[\![\mathcal{O}_T[\![P]\!]]\!]$ such that:

$$\delta = \langle OP, \rho_1, m_1\rangle^* \langle C'_1, \rho_1, m_1\rangle \langle OP, \rho_2, m_2\rangle^* ... \langle OP, \rho_n, m_n\rangle^* \langle C'_n, \rho_n, m_n\rangle$$

where $OP \in TrueOp(S)$, $act[\![C_i]\!] = act[\![C'_i]\!]$. Thus $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and $\mathcal{O}_T \in \mathbb{O}_c$. The same holds for the insertion of *false opaque predicates*.

An *unknown opaque predicate* $P^?$ sometimes evaluates to *true* and sometimes evaluates to *false*, thus the *true* and *false* branches have to exhibit equivalent behaviors. Usually, in order to avoid detection, the two branches present different obfuscated versions of the original command sequence. This can be seen as the composition of two or more distinct obfuscations: the first one $\mathcal{O}_U$ that inserts the unknown opaque predicates and duplicates the commands in such a way

that the two branches present the same code sequence, and subsequent ones that obfuscate the code in order to make the two branches look different. Let $\mathcal{O}_U : \mathbb{P} \to \mathbb{P}$ be the program transformation that inserts unknown opaque predicates, and let P, R, S and η be defined as in the code reordering case. In the considered programming language an unknown opaque predicate is expressed as $L : P^? \to \{L_T, L_F\}$. Let us define the set of commands encoding an unknown opaque predicate $P^?$ inserted by the transformation $\mathcal{O}_U$:

$$UnOp(S) \stackrel{\text{def}}{=} \left\{ L : P^? \to \{L_T, L_F\} \;\middle|\; \begin{array}{l} \exists C \in S : \\ lab[\![C]\!] = L, lab[\![\eta(C)]\!] = L_T \end{array} \right\}$$

$$Rep(UnOp(S)) \stackrel{\text{def}}{=} \left\{ R_1...R_k \;\middle|\; \begin{array}{l} R_1...R_k \in \wp(\mathbb{C}) \\ lab[\![R_1]\!] = L_F \end{array} \right\}$$

where $R_1...R_k$ present the same sequence of actions of the commands starting at label L_T. Then, $\mathcal{O}_U[\![P]\!] = (P \setminus S) \cup UnOp(S) \cup \eta(S) \cup Rep(UnOp(S))$. Observing the effects on program semantics we have that, for every trace $\sigma \in \mathbf{S}[\![P]\!]$, where $\sigma = \langle C_1, \rho_1, m_1\rangle ... \langle C_n, \rho_n, m_n\rangle$, there exists $\delta \in \mathbf{S}[\![\mathcal{O}_U[\![P]\!]]\!]$ such that:

$$\delta = \langle U, \rho_1, m_1\rangle^* \langle C'_1, \rho_1, m_1\rangle \langle U, \rho_2, m_2\rangle^* ... \langle U, \rho_n, m_n\rangle^* \langle C'_n, \rho_n, m_n\rangle$$

where $U \in UnOp(S)$ and $act[\![C_i]\!] = act[\![C'_i]\!]$. Thus $\alpha_e(\sigma) = \alpha_e(\delta)$, and $\mathcal{O}_U \in \mathbb{O}_c$.

Semantic NOP *insertion*

This transformation inserts commands that are irrelevant with respect to program trace semantics (see Fig. 6.3 for an example).

P

L_1	$: F := 1 \to L_2$
L_2	$: (X = 1) \to \{L_T, L_F\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: F := F \times X \to L_2$
L_T	$: ...$

$\mathcal{O}_N[\![P]\!]$

L_1	$: F := 1 \to L_2$
L_2	$: (X = 1) \to \{L_T, L_F\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: X := X \times 2 - X$
L_{F+2}	$: F := F \times X \to L_2$
L_T	$: ...$

Fig. 6.3. Semantic NOP insertion at label L_{F+1}

Let us consider $SN, C_1, C_2 \in \wp(\mathbb{C})$, SN is a semantic NOP with respect to $C_1 \cup C_2$ if for every $\sigma \in \mathbf{S}[\![C_1 \cup C_2]\!]$, there exists $\delta \in \mathbf{S}[\![C_1 \cup SN \cup C_2]\!]$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Let $\mathcal{O}_N : \mathbb{P} \to \mathbb{P}$ be the program transformation that inserts irrelevant instructions, therefore $\mathcal{O}_N[\![P]\!] = P \cup SN$ where SN represents the set of irrelevant instructions inserted in P. Following the definition of semantic

NOP we have that for every $\sigma \in \mathbf{S}[\![P]\!]$ there exists $\delta \in \mathbf{S}[\![\mathcal{O}_N[\![P]\!]]\!]$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$, thus $\mathcal{O}_N \in \mathbb{O}_C$.

Substitution of Equivalent Commands

This program transformation replaces a single command with an equivalent one, with the goal of thwarting signature matching (see Fig 6.4 for an example).

P	
L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

$\mathcal{O}_I[\![P]\!]$	
L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - X/X \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \times 2 - F \times X \rightarrow L_2$
L_T	$: ...$

Fig. 6.4. Substitution of equivalent commands at label L_F and L_{F+1}

Let $\mathcal{O}_I : \mathbb{P} \rightarrow \mathbb{P}$ be the program transformation that substitutes commands with equivalent ones. Two commands C and C' are equivalent if they always cause the same effects, namely if $\forall \xi \in \mathfrak{E}L\mathbf{C}[\![C]\!]\xi = \mathbf{C}[\![C']\!]\xi$. Thus, $\mathcal{O}_I[\![P]\!] = P'$ where $\forall C' \in P', \exists C \in P$ such that C and C' are equivalent. Observing the effects on program semantics we have that: for every $\sigma \in \mathbf{S}[\![P]\!]$ such that $\sigma = \langle C_1, \rho_1, m_1\rangle ... \langle C_n, \rho_n, m_n\rangle$, there exists $\delta \in \mathbf{S}[\![\mathcal{O}_J[\![P]\!]]\!]$ such that $\delta = \langle C'_1, \rho_1, m_1\rangle ... \langle C'_n, \rho_n, m_n\rangle$ where $\mathbf{C}\langle C_i, \rho_i, m_i\rangle = \mathbf{C}\langle C'_i, \rho_i, m_i\rangle$. Thus, $\alpha_e(\sigma) = \alpha_e(\delta)$, and $\mathcal{O}_I \in \mathbb{O}_c$.

Of course, malware writers usually combine different obfuscating transformations in order to prevent detection. The following result shows that the composition of conservative obfuscations is a conservative obfuscation. Thus, when more than one conservative obfuscation is applied, it can be handled as a single conservative obfuscation through abstraction α_c.

Lemma 6.10. Given $\mathcal{O}_1, \mathcal{O}_2 \in \mathbb{O}_c$ then $\mathcal{O}_1 \circ \mathcal{O}_2 \in \mathbb{O}_c$.

PROOF: By definition of conservative transformations we have that:

$$\forall \sigma \in \mathbf{S}[\![P]\!], \exists \delta \in \mathbf{S}[\![\mathcal{O}_1[\![P]\!]]\!] : \ \alpha_e(\sigma) \preceq \alpha_e(\delta)$$
$$\forall \delta \in \mathbf{S}[\![\mathcal{O}_1[\![P]\!]]\!], \exists \eta \in \mathbf{S}[\![\mathcal{O}_2[\![\mathcal{O}_1[\![P]\!]]\!]]\!] : \ \alpha_e(\delta) \preceq \alpha_e(\eta)$$

Thus for transitivity of $\preceq$: $\forall \sigma \in \mathbf{S}[\![P]\!], \exists \eta \in \mathbf{S}[\![\mathcal{O}_2[\![\mathcal{O}_1[\![P]\!]]\!]]\!]$ such that $\alpha_e(\sigma) \preceq \alpha_e(\eta)$, which proves that $\mathcal{O}_2 \circ \mathcal{O}_1$ is a conservative transformation.

□

Example 6.11. Let us consider a fragment of a malware M presenting the decryption loop used by polymorphic viruses. Such a fragment writes, starting from memory location B, the decryption of memory locations starting at location A and then executes the decrypted instructions. Observe that, given a variable X the semantics of $\pi_2(X)$ is the label expressed by $\pi_2(m(\rho(X)))$, in particular $\pi_2(n) = \bot$, while $\pi_2(A, S) = S$. Moreover, given a variable X, let $Dec(X)$ denote the execution of a set of commands that decrypt the value stored in the memory location $\rho(X)$. Let $\mathcal{O}_c[\![M]\!]$ be a conservative obfuscation of M obtained through code reordering, opaque predicate insertion and semantic NOP insertion.

M

L_1	$: \texttt{assign}(L_B, B) \rightarrow L_2$
L_2	$: \texttt{assign}(L_A, A) \rightarrow L_c$
L_c	$: cond(A) \rightarrow \{L_T, L_F\}$
L_T	$: B := Dec(A) \rightarrow L_{T_1}$
L_{T_1}	$: \texttt{assign}(\pi_2(B), B) \rightarrow L_{T_2}$
L_{T_2}	$: \texttt{assign}(\pi_2(A), A) \rightarrow L_C$
L_F	$: \texttt{skip} \rightarrow L_B$

$\mathcal{O}_c[\![M]\!]$

L_1	$: \texttt{assign}(L_B, B) \rightarrow L_2$
L_2	$: \texttt{skip} \rightarrow L_4$
L_c	$: cond(A) \rightarrow \{L_O, L_F\}$
L_4	$: \texttt{assign}(L_A, A) \rightarrow L_5$
L_5	$: \texttt{skip} \rightarrow L_c$
L_O	$: P^T \rightarrow \{L_N, L_k\}$
L_N	$: X := X - 3 \rightarrow L_{N_1}$
L_{N_1}	$: X := X + 3 \rightarrow L_T$
L_T	$: B := Dec(A) \rightarrow L_{T_1}$
L_{T_1}	$: \texttt{assign}(\pi_2(B), B) \rightarrow L_{T_2}$
L_{T_2}	$: \texttt{assign}(\pi_2(A), A) \rightarrow L_c$
L_k	$: \ldots$
L_F	$: \texttt{skip} \rightarrow L_B$

It can be shown that $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![\mathcal{O}_c[\![M]\!]]\!])) = \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!]))$, i.e., our semantics-based approach is able to see through the obfuscations and identify $\mathcal{O}[\![M]\!]$ as matching the malware M. In particular, let $\bot$ denote the undefined function.

$$
\begin{aligned}
\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!])) &= \alpha_e(\mathbf{S}[\![M]\!]) \\
&= (\bot, \bot), ((B \rightsquigarrow L_B), \bot), ((B \rightsquigarrow L_B, A \rightsquigarrow L_A), \bot)^2, \\
&\quad ((B \rightsquigarrow L_B, A \rightsquigarrow L_A), (\rho(B) \leftarrow \mathbf{Dec}(A))), \\
&\quad ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow L_A), (\rho(B) \leftarrow \mathbf{Dec}(A))), \\
&\quad ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow \pi_2(m(\rho(A))))), \\
&\quad (\rho(B) \leftarrow \mathbf{Dec}(A)))...
\end{aligned}
$$

while

$$\begin{aligned}\alpha_e(\mathbf{S}[\![\mathcal{O}_c[\![M]\!]]\!]) = {} & (\bot,\bot),((B \rightsquigarrow L_B),\bot)^2,((B \rightsquigarrow L_B, A \rightsquigarrow L_A),\bot)^5,\\ & ((B \rightsquigarrow L_B, A \rightsquigarrow L_A),(\rho(X) \leftarrow X-3)),\\ & ((B \rightsquigarrow L_B, A \rightsquigarrow L_A),(\rho(X) \leftarrow X+3, \rho(X) \leftarrow X-3)),\\ & ((B \rightsquigarrow L_B, A \rightsquigarrow L_A),(\rho(B) \leftarrow \mathbf{Dec}(A))),\\ & ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow L_A),(\rho(B) \leftarrow \mathbf{Dec}(A))),\\ & ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow \pi_2(m(\rho(A)))),(\rho(B) \leftarrow \mathbf{Dec}(A)))\\ & \ldots\end{aligned}$$

Thus, $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_c[\alpha_e(\mathbf{S}[\![M]\!])](\alpha_e(\mathbf{S}[\![\mathcal{O}_c[\![M]\!]]\!]))$.

□

6.4.2 Non-Conservative Obfuscations

An obfuscating transformation that does not satisfy the conservativeness condition is called *non-conservative*. A non-conservative transformation modifies the program semantics in such a way that the original environment-memory traces are not present in the semantics of the transformed program. A possible way to tackle these transformations is to identify the set of all possible modifications induced by a non-conservative obfuscation, and fix, when possible, a *canonical* one. In this way the abstraction would reduce the original semantics to the canonical version before checking malware infection. In the following we consider a non-conservative transformation, known as *variable renaming*, and propose a canonical abstraction that leads to a sound and complete semantic malware detector.

Another possible approach comes from Theorem 6.7 that states that if α is preserved by $\mathcal{O}$ then the semantic malware detector on α is complete with respect to $\mathcal{O}$. Recall that, given a program transformation $\mathcal{O} : \mathbb{P} \to \mathbb{P}$, it is possible to systematically derive the most concrete abstraction preserved by $\mathcal{O}$, as shown in Chapter 4. This systematic methodology can be used in presence of non-conservative obfuscations in order to derive a complete semantic malware detector when it is not easy to identify a canonical abstraction.

Moreover, in Section 6.5 we show how it is possible to handle a class of non-conservative obfuscations through a further abstraction of the malware semantics.

Variable Renaming

Variable renaming is a simple obfuscating transformation, often used to prevent signature matching, that replaces the names of variables with some different new names (see Fig. 6.5 for an example).

P	
L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

$\mathcal{O}_J[\![P]\!]$	
L_1	$: P := 1 \rightarrow L_2$
L_2	$: (Y = 1) \rightarrow \{L_T, L_F\}$
L_F	$: Y := Y - 1 \rightarrow L_{F+1}$
L_{F+1}	$: P := P \times Y \rightarrow L_2$
L_T	$: ...$

Fig. 6.5. Variable renaming

Assuming that every environment function associates variable V_L to memory location L, allows us to reason about variable renaming also in the case of compiled code, where variable names have disappeared. Let $\mathcal{O}_v : \mathbb{P} \times \Pi \rightarrow \mathbb{P}$ denote the obfuscating transformation that, given a program P, renames its variables according to a mapping $\pi \in \Pi$, where $\pi : \mathit{var}[\![P]\!] \rightarrow \mathit{Names}$ is a bijective function that relates the name of each program variable in $\mathit{var}[\![P]\!]$ to its new name in Names.

$$\mathcal{O}_v(P, \pi) \stackrel{\text{def}}{=} \left\{ C \;\middle|\; \begin{array}{l} \exists C' \in P : \mathit{lab}[\![C]\!] = \mathit{lab}[\![C']\!], \\ \mathit{suc}[\![C]\!] = \mathit{suc}[\![C']\!], \\ \mathit{act}[\![C]\!] = \mathit{act}[\![C']\!][X/\pi(X)] \end{array} \right\}$$

where $A[X/\pi(X)]$ represents action A where each variable name X is replaced by the new name $\pi(X)$. Recall that the matching relation between program traces considers the abstraction α_e of traces, thus it is interesting to observe that:

$$\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![P, \pi]\!]]\!]) = \alpha_v[\pi](\alpha_e(\mathbf{S}[\![P]\!])) \tag{6.2}$$

where $\alpha_v : \Pi \rightarrow (\mathfrak{X}^* \rightarrow \mathfrak{X}^*)$ is defined as:

$$\alpha_v[\pi]((\rho_1, m_1) \ldots (\rho_n, m_n)) \stackrel{\text{def}}{=} (\rho_1 \circ \pi^{-1}, m_1) \ldots (\rho_n \circ \pi^{-1}, m_n)$$

In order to deal with variable renaming obfuscation we introduce the notion of *canonical variable renaming*, denoted as $\widehat{\pi}$. The idea of canonical mappings is that there exists a renaming $\pi : \mathit{var}[\![P]\!] \rightarrow \mathit{var}[\![Q]\!]$ that transforms program P into program Q, namely such that $\mathcal{O}_v[\![P, \pi]\!] = Q$, if and only if $\alpha_v[\widehat{\pi}](\alpha_e(\mathbf{S}[\![Q]\!])) = \alpha_v[\widehat{\pi}](\alpha_e(\mathbf{S}[\![P]\!]))$. This means that a program Q is a renamed version of program P if and only if Q and P are indistinguishable after canonical renaming. In the following we define a possible canonical renaming for the variables of a given a program.

Let $\{V_i\}_{i \in \mathbb{N}}$ be a set of canonical variable names. The idea is to order the variables appearing in program semantics $\mathbf{S}[\![P]\!]$, and to define a canonical renaming that renames the first variable with V_1, the second with V_2 and so on. The set $\mathbb{L}$ of memory locations is an ordered set with ordering relation $\leq_L$. With

a slight abuse of notation we denote with $\leq_L$ also the lexicographical order induced by $\leq_L$ on sequences of memory locations. Let us define the ordering $\leq_\Sigma$ over traces Σ^* where, given $\sigma, \delta \in \Sigma^*$:

$$\sigma \leq_\Sigma \delta \quad \text{if} \quad \begin{cases} |\sigma| \leq |\delta| \text{ or} \\ |\sigma| = |\delta| \text{ and } lab(\sigma_1)lab(\sigma_2)...lab(\sigma_n) \leq_L lab(\delta_1)lab(\delta_2)...lab(\delta_n) \end{cases}$$

where $lab(\langle C, \rho, m\rangle) = lab[\![C]\!]$. It is clear that, given a program P, the ordering $\leq_\Sigma$ on its traces induces an order on the set $\mathcal{Z} = \alpha_e(\mathbf{S}[\![P]\!])$ of its environment-memory traces, i.e., given $\sigma, \delta \in \mathbf{S}[\![P]\!]$:

$$\sigma \leq_\Sigma \delta \quad \Rightarrow \quad \alpha_e(\sigma) \leq_\mathcal{Z} \alpha_e(\delta)$$

By definition, the set of variables assigned in $\mathcal{Z}$ is exactly $var[\![P]\!]$, therefore, for equation (6.2), a canonical renaming $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$, is such that $\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![P, \widehat{\pi}_P]\!]]\!]) = \alpha_v[\widehat{\pi}_P](\mathcal{Z})$. Let $\bar{\mathcal{Z}}$ denote the list of environment-memory traces of $\mathcal{Z} = \alpha_e(\mathbf{S}[\![P]\!])$ ordered following the ordering $\leq_\mathcal{Z}$ defined above. Let B be a list, then $hd(B)$ returns the first element of the list, $tl(B)$ returns list B without the first element, $B : e$ ($e : B$) is the list resulting by inserting element e at the end (beginning) of B, $B[i]$ returns the i-th element of the list, and $e \in B$ means that e is an element of B. The relation $\leq_\mathcal{Z}$ defines an order between context traces in $\alpha_e(\mathbf{S}[\![P]\!])$, now we need to define an order between the variables in a context trace. Given $s \in \mathfrak{X}^*$, the idea is to order the variables according to their assignment time. Note that, program execution starts from the uninitialized environment $\rho_{uninit} = \lambda X.\bot$, and that each command assigns at most one variable. Let $def(\rho)$ denote the set of variables that have defined (i.e., non-$\bot$) values in an environment ρ. This means that considering $s \in \mathfrak{X}^*$ we have that $def(\rho_{i-1}) \subseteq def(\rho_i)$, and if $def(\rho_{i-1}) \subset def(\rho_i)$ then $def(\rho_i) = def(\rho_{i-1}) \cup \{X\}$ where $X \in \mathbb{X}$ is the new variable assigned to memory location $\rho_i(X)$. Given $s \in \mathfrak{X}^*$, let us define $List(s)$ as the list of variables in s ordered according to their assignment time. Formally, let $s = (\rho_1, m_1)(\rho_2, m_2)...(\rho_n, m_n) = (\rho_1, m_1)s'$:

$$List(s) = \begin{cases} \epsilon & \text{if } s = \epsilon \\ X : List(s') & \text{if } def(s_2) \setminus def(s_1) = \{X\} \\ List(s') & \text{if } def(s_2) \setminus def(s_1) = \varnothing \end{cases}$$

Algorithm 1, given a list $\bar{\mathcal{Z}}$ encoding the ordering $\leq_\mathcal{Z}$ on context traces in $\alpha_e(\mathbf{S}[\![P]\!])$, and given $List(s)$ for every $s \in \alpha_e(\mathbf{S}[\![P]\!])$ encoding the assignment ordering of variables in s, returns the list $Rename[\mathcal{Z}]$ encoding the ordering of variables in $\alpha_e(\mathbf{S}[\![P]\!])$. Given $\mathcal{Z} = \alpha_e(\mathbf{S}[\![P]\!])$, we rename its variables following the canonical renaming $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$ that associates the new canonical name V_i to the variable of P in the i-th position in the list $Rename[\mathcal{Z}]$. Thus, the canonical renaming $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$ is defined as follows:

Input: A list of context sequences $\bar{\mathcal{Z}}$, with $\mathcal{Z} \in \alpha_e(\mathbf{S}[\![P]\!])$.
Output: A list $Rename[\mathcal{Z}]$ used to associate canonical variable V_i to the variable in the list position i.

```
Rename[Z] = List(hd(Z̄))
Z̄ = tl(Z̄)
while (Z̄ ≠ ∅) do
    trace = List(hd(Z̄))
    while (trace ≠ ∅) do
        if (hd(trace) ∉ Rename[Z]) then
            Rename[Z] = Rename[Z] : hd(trace)
        end
        trace = tl(trace)
    end
    Z̄ = tl(Z̄)
end
```

Algorithm 1: Algorithm for canonical renaming of variables.

$$\widehat{\pi}_P(X) = V_i \Leftrightarrow Rename[\mathcal{Z}][i] = X$$

The following result is necessary in order to prove that the mapping $\widehat{\pi}_P$ defined above is a canonical renaming.

Lemma 6.12. Given two programs $P, Q \in \mathbb{P}$ let $\mathcal{Z} = \alpha_e(\mathbf{S}[\![P]\!])$ and $\mathcal{Y} = \alpha_e(\mathbf{S}[\![Q]\!])$. Then we have that:

1 $\alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\mathcal{Y}) \quad \Rightarrow \quad \exists \pi : var[\![P]\!] \to var[\![Q]\!] : \alpha_v[\pi](\mathcal{Z}) = \mathcal{Y}$
2 $\exists \pi : var[\![P]\!] \to var[\![Q]\!] : \alpha_v[\pi](\mathcal{Z}) = \mathcal{Y}$ and $(\alpha_v[\pi](s) = t \Rightarrow (\bar{\mathcal{Z}}[i] = s \wedge \mathcal{Y}[i] = t)) \quad \Rightarrow \quad \alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\mathcal{Y})$

PROOF:

1 Assume $\alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\mathcal{Y})$, i.e., $\{\alpha_v[\widehat{\pi}_P](s) \mid s \in \mathcal{Z}\} = \{\alpha_v[\widehat{\pi}_Q](t) \mid t \in \mathcal{Y}\}$. This means that $|var[\![\mathcal{Z}]\!]| = |var[\![\mathcal{Y}]\!]| = k$, and that $\widehat{\pi}_P : var[\![\mathcal{Z}]\!] \to \{V_1 ... V_k\}$ while $\widehat{\pi}_Q : var[\![\mathcal{Y}]\!] \to \{V_1 ... V_k\}$. Recall that $var[\![\mathcal{Z}]\!] = var[\![P]\!]$ and $var[\![\mathcal{Y}]\!] = var[\![Q]\!]$. Let us define $\pi : var[\![P]\!] \to var[\![Q]\!]$ as $\pi \stackrel{\text{def}}{=} \widehat{\pi}_Q^{-1} \circ \widehat{\pi}_P$. The mapping π is bijective since it is obtained as composition of bijective functions. Let us show that π satisfies the condition on the left, namely that $\mathcal{Y} = \alpha_v[\pi](\mathcal{Z})$. To prove this we show that given $s \in \mathcal{Z}$ and $t \in \mathcal{Y}$ such that $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](t)$ then $\alpha_v[\pi](s) = t$. Let $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](t) = (\widehat{\rho}_1, m_1)...(\widehat{\rho}_n, m_n)$, while $s = (\rho_1^s, m_1)...(\rho_n^s, m_n)$ and $t = (\rho_1^t, m_1)...(\rho_n^t, m_n)$. Then:

$$\begin{aligned}
\alpha_v[\pi](s) &= (\rho_1^s \circ \pi^{-1}, m_1)...(\rho_n^s \circ \pi^{-1}, m_n) \\
&= (\rho_1^s \circ \widehat{\pi}_P^{-1} \circ \widehat{\pi}_Q, m_1)...(\rho_n^s \circ \widehat{\pi}_P^{-1} \circ \widehat{\pi}_Q, m_n) \\
&= (\widehat{\rho}_1 \circ \widehat{\pi}_Q, m_1)...(\widehat{\rho}_n \circ \widehat{\pi}_Q, m_n) \\
&= (\rho_1^t, m_1)...(\rho_n^t, m_n) = t
\end{aligned}$$

2 Assume $\exists \pi : var[\![P]\!] \to var[\![Q]\!]$ such that $\mathcal{Y} = \alpha_v[\pi](\mathcal{Z})$. By definition $\mathcal{Y} = \{\alpha_v[\pi](s) \mid s \in \mathcal{Z}\}$. Let us show that $\alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\{\alpha_v[\pi](s) \mid s \in \mathcal{Z}\})$. We prove this by showing that $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](\alpha_v[\pi](s))$. By definition we have that $|\mathcal{Y}| = |\mathcal{Z}|$ and $|var[\![P]\!]| = |var[\![Q]\!]| = k$, moreover we have $\pi : var[\![P]\!] \to var[\![Q]\!]$. Given $s \in \mathcal{Z}$ and $t \in \mathcal{Y}$ such that $t = \alpha_v[\pi](s)$ then $|s| = |t|$ and $|var[\![s]\!]| = |var[\![t]\!]|$, thus $List(s)[i] = X$ and $List(t)[i] = \pi(X)$, moreover, by hypothesis, $\widetilde{\mathcal{Z}}[i] = s$ and $\widetilde{\mathcal{Y}}[i] = t$. This hold for every pair of traces obtained trough renaming. Therefore, considering the canonical rename for $\mathcal{Y}$ as given by $\widehat{\pi}_Q \stackrel{\text{def}}{=} \widehat{\pi}_P \circ \pi^{-1}$, we have that $\forall s \in \mathcal{Z}, t \in \mathcal{Y}$ such that $\alpha_v[\pi](s) = t$ then $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](t)$. In fact:

$$\begin{aligned}
\alpha_v[\widehat{\pi}_Q](t) &= \alpha_v[\widehat{\pi}_Q](\alpha_v[\pi](s)) \\
&= \alpha_v[\widehat{\pi}_Q]((\rho_1^s \circ \pi^{-1}, m_1)...(\rho_n^s \circ \pi^{-1}, m_n)) \\
&= (\rho_1^s \circ \pi^{-1} \circ \widehat{\pi}_Q^{-1}, m_1)...(\rho_n^s \circ \pi^{-1} \circ \widehat{\pi}_Q^{-1}, m_n) \\
&= (\rho_1^s \circ \pi^{-1} \circ \pi \circ \widehat{\pi}_P^{-1}, m_1)...(\rho_n^s \circ \pi^{-1} \circ \pi \circ \widehat{\pi}_P^{-1}, m_n) \\
&= (\rho_1^s \circ \widehat{\pi}_P^{-1}, m_1)...(\rho_n^s \circ \widehat{\pi}_P^{-1}, m_n) \\
&= (\widehat{\rho}_1, m_1)...(\widehat{\rho}_n, m_n) = \alpha_v[\widehat{\pi}_P](s)
\end{aligned}$$

□

Let $\widehat{\Pi}$ denote a set of canonical variable renamings, the additive function $\alpha_v : \widehat{\Pi} \to (\wp(\mathfrak{X}^*) \to \wp(\mathfrak{X}_c^*))$, where $\mathfrak{X}_c$ denotes execution contexts where environments are defined on canonical variables, is an approximation that abstracts from the names of variables. Thus, we have the following Galois connection:

$$\langle \wp(\mathfrak{X}^*), \subseteq \rangle \underset{\alpha_v[\widehat{\Pi}]}{\overset{\gamma_v[\widehat{\Pi}]}{\leftrightarrows}} \langle \wp(\mathfrak{X}_c^*), \subseteq \rangle$$

The following result, where $\widehat{\pi}_M$ and $\widehat{\pi}_{P_r}$ denote respectively the canonical renaming of the malware variables and of restricted program variables, shows that the semantic malware detector on $\alpha_v[\widehat{\Pi}]$ is both complete and sound for variable renaming.

Theorem 6.13. *$\exists \pi : \mathcal{O}_v[\![M, \pi]\!] \hookrightarrow P$ if and only if*

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_v[\widehat{\pi}_M](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$$

PROOF: ($\Rightarrow$) Completeness: Assume that $\mathcal{O}_v[\![M, \pi]\!] \hookrightarrow P$, this means that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $P_r = \mathcal{O}_v[\![M, \pi]\!]$. Therefore $\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])) = \alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![M, \pi]\!]]\!])$. Thus, in order to conclude the proof we have to show that $\alpha_v[\widehat{\pi}_M](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![M, \pi]\!]]\!]))$. Recall that $\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![M, \pi]\!]]\!]) = \alpha_v[\pi](\alpha_e(\mathbf{S}[\![M]\!]))$. Following Lemma 6.12 point 2 we have that:

$$\alpha_v[\widehat{\pi}_M](\alpha_e(\mathbf{S}[\![M]\!])) = \alpha_v[\widehat{\pi}_{P_r}](\alpha_v[\pi](\alpha_e(\mathbf{S}[\![M]\!]))) = \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![M,\pi]\!]]\!]))$$

which concludes the proof.
($\Leftarrow$) Soundness: Assume that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_v[\widehat{\pi}_M](\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. Let α_R be the program restriction that satisfies the above equation with equality: $\alpha_v[\widehat{\pi}_M](\alpha_e(\mathbf{S}[\![M]\!])) = \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\alpha_R(\mathbf{S}[\![P]\!])))$. It is clear that $\alpha_R(\mathbf{S}[\![P]\!]) \subseteq \alpha_r(\mathbf{S}[\![P]\!])$. From Lemma 6.12 point 1 we have that $\exists \pi : var[\![M]\!] \to var[\![P_R]\!]$ such that $\alpha_e(\alpha_R(\mathbf{S}[\![P]\!])) = \alpha_v[\pi](\alpha_e(\mathbf{S}[\![M]\!])) = \alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![M,\pi]\!]]\!])$, namely $\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![M,\pi]\!]]\!]) = \alpha_e(\alpha_R(\mathbf{S}[\![P]\!])) \subseteq \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$, meaning that $\mathcal{O}_v[\![M,\pi]\!] \hookrightarrow P$.

□

6.4.3 Composition

As observed earlier, in general, malware writers use multiple obfuscating transformations concurrently to prevent detection, therefore we have to consider the composition of non-conservative obfuscations (Lemma 6.10 regards composition of conservative obfuscations only). Investigating the relation between abstractions α_1 and α_2, on which the semantic malware detector is complete (resp. sound) respectively for obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$, and the abstraction that is complete (resp. sound) for their compositions, i.e., for $\{\mathcal{O}_1 \circ \mathcal{O}_2, \mathcal{O}_2 \circ \mathcal{O}_1\}$, we have obtained the following result.

Theorem 6.14. Given two abstractions α_1 and α_2 and two obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$ then:

1 if the semantic malware detector on α_1 is complete for $\mathcal{O}_1$, the semantic malware detector on α_2 is complete for $\mathcal{O}_2$, and $\alpha_1 \circ \alpha_2 = \alpha_2 \circ \alpha_1$, then the semantic malware detector on $\alpha_1 \circ \alpha_2$ is complete for $\{\mathcal{O}_1 \circ \mathcal{O}_2, \mathcal{O}_2 \circ \mathcal{O}_1\}$;
2 if the semantic malware detector on α_1 is sound for $\mathcal{O}_1$, the semantic malware detector on α_2 is sound for $\mathcal{O}_2$, and $\alpha_1(X) \subseteq \alpha_1(Y) \Rightarrow X \subseteq Y$, then the semantic malware detector on $\alpha_1 \circ \alpha_2$ is sound for $\mathcal{O}_1 \circ \mathcal{O}_2$.

PROOF:

1 Recall that the semantic malware detector on α_i is complete for $\mathcal{O}_i$ if $\mathcal{O}_i[\![M]\!] \hookrightarrow P \Rightarrow \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_i(\alpha_e(\mathbf{S}[\![P]\!])) \subseteq \alpha_i(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. Assume that $\mathcal{O}_1[\![\mathcal{O}_2[\![P]\!]]\!] \hookrightarrow P$, this means that there exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![\mathcal{O}_1[\![\mathcal{O}_2[\![P]\!]]\!]]\!] = \alpha_r(\mathbf{S}[\![P]\!])$. Since the semantic malware detector on α_1 is complete for $\mathcal{O}_1$, we have that: $\alpha_1(\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!])) \subseteq \alpha_1(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. Abstraction α_2 is monotone and therefore:

$$\alpha_2(\alpha_1(\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!]))) \subseteq \alpha_2(\alpha_1(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$$

In general we have that $\mathcal{O}_2[\![M]\!] \hookrightarrow \mathcal{O}_2[\![M]\!]$, and since α_2 is complete we have that $\alpha_2(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_2(\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!]))$. Abstraction α_1 is monotone and therefore $\alpha_1(\alpha_2(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_1(\alpha_2(\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!])))$. Since α_1 and α_2 commute we have:

$$\alpha_2(\alpha_1(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_2(\alpha_1(\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!])))$$

Thus, $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_1(\alpha_2(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_2(\alpha_1(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$. The proof that $\mathcal{O}_2[\![\mathcal{O}_1[\![M]\!]]\!] \hookrightarrow P$ implies that there exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_1(\alpha_2(\alpha_e\mathbf{S}[\![M]\!])) \subseteq \alpha_1(\alpha_2(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$ is analogous.

2 We have to prove that if $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha_1(\alpha_2(\alpha_e(\mathbf{S}[\![P]\!]))) \subseteq \alpha_1(\alpha_2(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$ then $\mathcal{O}_1[\![\mathcal{O}_2[\![M]\!]]\!] \hookrightarrow P$.
Assume $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_1(\alpha_2(\alpha_e(\mathbf{S}[\![P]\!]))) \subseteq \alpha_1(\alpha_2(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$, since $\alpha_1(X) \subseteq \alpha_1(Y) \Rightarrow X \subseteq Y$ we have that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha_2(\alpha_e(\mathbf{S}[\![P]\!])) \subseteq \alpha_2(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. The semantic malware detector on α_2 is sound by hypothesis, therefore $\mathcal{O}_2[\![M]\!] \hookrightarrow P$, namely $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!]) \subseteq \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$. Abstraction α_1 is monotone and therefore $\alpha_1(\alpha_e(\mathbf{S}[\![\mathcal{O}_2[\![M]\!]]\!])) \subseteq \alpha_1(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. The semantic malware detector on α_1 is sound by hypothesis and therefore $\mathcal{O}_1[\![\mathcal{O}_2[\![M]\!]]\!] \hookrightarrow P$.

□

Thus, in order to propagate completeness through composition $\mathcal{O}_1 \circ \mathcal{O}_2$ and $\mathcal{O}_2 \circ \mathcal{O}_1$ the corresponding abstractions have to be independent. On the other side, in order to propagate soundness through composition $\mathcal{O}_1 \circ \mathcal{O}_2$ the abstraction α_1, corresponding to the last applied obfuscation, has to be an order-embedding, namely α_1 has to be both order-preserving and order-reflecting, i.e., $\alpha_1(X) \subseteq \alpha_1(Y) \Leftrightarrow X \subseteq Y$. Observe that, when composing a non-conservative obfuscation $\mathcal{O}$, for which the semantic malware detector on $\alpha_{\mathcal{O}}$ is complete, with a conservative obfuscation $\mathcal{O}_c$, the commutation condition $\alpha_{\mathcal{O}} \circ \alpha_c = \alpha_c \circ \alpha_{\mathcal{O}}$ of point 1 of the above theorem is satisfied if and only if $(\alpha_e(\sigma) \preceq \alpha_e(\delta)) \Leftrightarrow \alpha_{\mathcal{O}}(\alpha_e(\sigma)) \preceq \alpha_{\mathcal{O}}(\alpha_e(\delta))$. In fact, only in this case α_c and $\alpha_{\mathcal{O}}$ commute, as shown by the following equations:

$$\begin{aligned}
\alpha_{\mathcal{O}}(\alpha_c[S](\alpha_e(\sigma))) &= \alpha_{\mathcal{O}}(S \cap \mathit{Subseq}(\alpha_e(\sigma))) \\
&= \{\ \alpha_{\mathcal{O}}(\alpha_e(\delta)) \,\big|\, \alpha_e(\delta) \in S \cap \mathit{SubSeq}(\alpha_e(\sigma))\ \} \\
&= \alpha_{\mathcal{O}}(S) \cap \{\ \alpha_{\mathcal{O}}(\alpha_e(\delta)) \,\big|\, \alpha_e(\delta \preceq \alpha_e(\sigma))\ \}
\end{aligned}$$

$$\begin{aligned}
\alpha_c[\alpha_{\mathcal{O}}(S)](\alpha_{\mathcal{O}}(\alpha_e(\sigma))) &= \alpha_{\mathcal{O}}(S) \cap \mathit{SubSeq}(\alpha_{\mathcal{O}}(\alpha_e(\sigma))) \\
&= \alpha_{\mathcal{O}}(S) \cap \{\ \alpha_{\mathcal{O}}(\alpha_e(\delta)) \,\big|\, \alpha_{\mathcal{O}}(\alpha_e(\delta)) \preceq \alpha_{\mathcal{O}}(\alpha_e(\sigma))\ \}
\end{aligned}$$

Example 6.15. Let us consider $\mathcal{O}_v[\![\mathcal{O}_c[\![M]\!], \pi]\!]$ obtained by obfuscating the portion of malware M in Example 6.11 through variable renaming and some conservative obfuscations, where the renaming function is defined by $\pi(B) = D, \pi(A) = E$. It is clear that variable renaming preserves $\preceq$, namely $\alpha_v[\pi]\alpha_e(\sigma) \preceq \alpha_v[\pi]\alpha_e(\delta)$ if and only if $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. In fact, it is possible to show that:

$$\alpha_c[\alpha_v[\widehat{\Pi}](\alpha_e(\mathbf{S}[\![M]\!])](\alpha_v[\widehat{\Pi}](\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_c[\alpha_v[\widehat{\Pi}](\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_v[\widehat{\Pi}](\alpha_e(\alpha_r(\mathbf{S}[\![\mathcal{O}_v[\![\mathcal{O}_c[\![M]\!], \pi]\!]]\!]))))$$

$\mathcal{O}_v[\![\mathcal{O}_c[\![M]\!], \pi]\!]$

$L_1 : \texttt{assign}(D, L_B) \rightarrow L_2$
$L_2 : \texttt{skip} \rightarrow L_4$
$L_c : cond(E) \rightarrow \{L_O, L_F\}$
$L_4 : \texttt{assign}(E, L_A) \rightarrow L_5$
$L_5 : \texttt{skip} \rightarrow L_c$
$L_O : P^T \rightarrow \{L_T, L_k\}$
$L_T : D := Dec(E) \rightarrow L_{T_1}$
$L_{T_1} : \texttt{assign}(\pi_2(D), D) \rightarrow L_{T_2}$
$L_{T_2} : \texttt{assign}(\pi_2(E), E) \rightarrow L_c$
$L_k : \ldots$
$L_F : \ldots$

Namely, given the abstractions $\alpha_c[\alpha_e(\mathbf{S}[\![M]\!])]$ and α_v on which, by definition, the semantic malware detector is complete respectively for $\mathcal{O}_c$ and $\mathcal{O}_v$, the semantic malware detector on $\alpha_c \circ \alpha_v$ is complete for the composition $\mathcal{O}_v \circ \mathcal{O}_c$. Let $\bot$ denote the undefined function, then we have the following.
$\alpha_c[\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![M]\!])](\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![M]\!])) =$

$$(\bot, \bot), ((V_1 \rightsquigarrow L_B), \bot), ((V_1 \rightsquigarrow L_B, V_2 \rightsquigarrow L_A), \bot)^2, ((V_1 \rightsquigarrow L_B, V_2 \rightsquigarrow L_A), (\rho(V_1) \leftarrow \mathbf{Dec}(V_2))), ((V_1 \rightsquigarrow \pi_2(m(\rho(V_1))), V_2 \rightsquigarrow L_A), (\rho(V_1) \leftarrow \mathbf{Dec}(V_2))), ((V_1 \rightsquigarrow \pi_2(m(\rho(V_1)), V_2 \rightsquigarrow \pi_2(m(\rho(V_2)))), (\rho(V_1) \leftarrow \mathbf{Dec}(V_2))), \ldots$$

while, $\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![\mathcal{O}_c[\![M, \pi]\!]]\!]]\!]) =$

$$(\bot, \bot), ((V_1 \rightsquigarrow L_B), \bot)^2, ((V_1 \rightsquigarrow L_B, V_2 \rightsquigarrow L_A), \bot)^5, ((V_1 \rightsquigarrow L_B, V_2 \rightsquigarrow L_A), (\rho(V_1) \leftarrow \mathbf{Dec}(V_2)))((V_1 \rightsquigarrow \pi_2(m(\rho(V_1)), V_2 \rightsquigarrow L_A), (\rho(V_1) \leftarrow \mathbf{Dec}(V_2))), ((V_1 \rightsquigarrow \pi_2(m(\rho(V_1)), V_2 \rightsquigarrow \pi_2(m(\rho(V_2)))), (\rho(V_1) \leftarrow \mathbf{Dec}(V_2))), \ldots$$

Thus:
$\alpha_c[\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![M]\!])](\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![\mathcal{O}_v[\![\mathcal{O}_c[\![M]\!], \pi]\!]]\!])) =$
$\alpha_c[\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![M]\!])](\alpha_v[\widehat{\Pi}]\alpha_e(\mathbf{S}[\![M]\!]))$

□

6.5 Further Malware Abstractions

Definition 6.4 characterizes the presence of a malware M in a program P as the existence of a restriction $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha_e(\mathbf{S}[\![M]\!]) \subseteq \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$. This means that program P is infected by malware M if for every malware behaviour there exists a program behaviour that matches it. In the following we show how this notion of malware infection can be weakened in three different ways. First, we can abstract the malware traces by eliminating the states that are not relevant to determine maliciousness, and then check if program P matches this simplified behavior (i.e., interesting states). Second, we can require program P to match a proper subset of malicious behaviors (i.e., interesting behaviours). Furthermore these two notions of malware infection can be combined by requiring program P to match some states on a subset of malware behaviors. Finally, the infection condition can be expressed in terms of a sequence of actions rather than a sequence representing the evolution of the execution context (i.e., interesting actions). Once again, action abstraction can be combined with either states abstraction or behaviours abstraction or with both of them. It is clear that a deeper understanding of the malware behavior is necessary in order to specify each of the proposed simplifications.

6.5.1 Interesting States

The maliciousness of a malware behaviour may be expressed by the fact that some (malware) states are reached in a certain order during program execution. Observe that this condition is clearly implied by, i.e., weaker than, the (standard) matching relation between all malware traces and the restricted program traces. Let us use the *interesting states* of a malware to refer to those states that capture the malicious behaviour. Assume that we have an oracle that, given a malware M, returns the set of its interesting states $Int(M) \subseteq \Sigma[\![M]\!]$. These states could be selected based on a security policy. For example, the states could represent the result of network operations. This means that in order to verify if program P is infected by malware M, we have to check whether the malicious sequences of interesting states are present in P. Let us define the trace transformation $\alpha_{Int(M)} : \mathfrak{X}^* \to \mathfrak{X}^*$ which considers only the interesting contexts in a given trace $s = \xi_1 s'$:

$$\alpha_{Int(M)}(s) = \begin{cases} \epsilon & \text{if } s = \epsilon \\ \xi_1 \alpha_{Int(M)}(s') & \text{if } \xi_1 \in \alpha_e(Int(M)) \\ \alpha_{Int(M)}(s') & \text{otherwise} \end{cases}$$

The following definition characterizes the presence of malware M in terms of its interesting states, i.e., through abstraction $\alpha_{Int(M)}$.

Definition 6.16. A program P is infected by a vanilla malware M with interesting states $Int(M)$, i.e., $M \hookrightarrow_{Int(M)} P$, if $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:

$$\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!])) \subseteq \alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$$

Thus, we can weaken the standard notion of conservative transformation by saying that $\mathcal{O} : \mathbb{P} \to \mathbb{P}$ is *conservative with respect to* $Int(M)$ if for every malware trace $\sigma \in \mathbf{S}[\![M]\!]$ there exists a program trace $\delta \in \mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]$ such that $\alpha_{Int(M)}(\alpha_e(\sigma)) \preceq \alpha_{Int(M)}(\alpha_e(\delta))$

When program infection is characterized by Definition 6.16, the semantic malware detector on $\alpha_c \circ \alpha_{Int(M)}$ is complete and sound for the obfuscating transformations that are conservative with respect to $Int(M)$.

Theorem 6.17. Let $Int(M)$ be the set of interesting states of a vanilla malware M. Then we have that:
Completeness: For every obfuscation $\mathcal{O}$ which is conservative with respect to $Int(M)$, if $\mathcal{O}[\![M]\!] \hookrightarrow_{Int(M)} P$ there exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:

$$\alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$$

Soundness: If there exists $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:

$$\alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$$

then there exists an obfuscation $\mathcal{O}$ that is conservative with respect to $Int(M)$ such that $\mathcal{O}[\![M]\!] \hookrightarrow P$.

PROOF: Completeness: Let $\mathcal{O}$ be a conservative obfuscation with respect to $Int(M)$ such that $\mathcal{O}[\![M]\!] \hookrightarrow_{Int(M)} P$, then it means that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $P_r = \mathcal{O}[\![M]\!]$, namely $\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!])) = \alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$. Therefore, we have that:

$$\alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]))) = \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$$

Thus, we have to show that:

$$\alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(S[\![\mathcal{O}[\![M]\!]]\!])))$$

By hypothesis $\mathcal{O}$ is conservative with respect to $Int(M)$, thus we have that for every $\sigma \in \mathbf{S}[\![M]\!]$, there exists $\delta \in \mathbf{S}[\![\mathcal{O}[\![M]\!]]\!] : \alpha_{Int(M)}(\alpha_e(\sigma)) \preceq \alpha_{Int(M)}(\alpha_e(\delta))$. Moreover, for every $s \in \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!])))$ there exists $\sigma \in \mathbf{S}[\![M]\!] : s = \alpha_{Int(M)}(\alpha_e(\sigma))$, therefore $\forall \sigma \in \mathbf{S}[\![M]\!]$, there exists $\delta \in$

$\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]$ such that $s = \alpha_{Int(M)}(\alpha_e(\sigma)) \preceq \alpha_{Int(M)}(\alpha_e(\delta))$, and $\alpha_{Int(M)}(\alpha_e(\delta)) = t \in \alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]))$. This means that $\forall s \in \alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!])), \exists t \in \alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]))$ such that $s \in SubSeq(t)$. Hence, $\forall s \in \alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))$ we have that

$$s \in \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!])))$$

which concludes the proof.
Soundness: Assume that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:

$$\alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))) \subseteq \alpha_c[\alpha_{Int(M)}(\alpha_e(S[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$$

This means that $\forall \sigma \in \mathbf{S}[\![M]\!]$:

$$\alpha_{Int(M)}(\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!]))](\alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))))$$

and for every $\sigma \in \mathbf{S}[\![M]\!]$ there exists $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$ such that $\alpha_{Int(M)}(\alpha_e(\sigma)) \in \alpha_{Int(M)}(\alpha_e(\mathbf{S}[\![M]\!])) \cap SubSeq(\alpha_{Int(M)}(\alpha_e(\delta)))$. This means that $\forall \sigma \in \mathbf{S}[\![M]\!]$ there exists $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$ such that $\alpha_{Int(M)}(\alpha_e(\sigma)) \preceq \alpha_{Int(M)}(\alpha_e(\delta))$, which means that P_r is a conservative obfuscation of M with respect to $Int(M)$.

□

It is clear that transformations that are non-conservative may be conservative with respect to $Int(M)$, meaning that knowing the set of interesting states of a malware allows us to handle also some non-conservative obfuscations. For example the abstraction $\alpha_{Int(M)}$ may allow the semantic malware detector to deal with reordering of independent instructions, as the following example shows.

Example 6.18. Let us consider the malware M and its obfuscation $\mathcal{O}[\![M]\!]$ obtained by reordering independent instructions.

M
$L_1 : A_1 \rightarrow L_2$
$L_2 : A_2 \rightarrow L_3$
$L_3 : A_3 \rightarrow L_4$
$L_4 : A_4 \rightarrow L_5$
$L_5 : A_5 \rightarrow L_6$

$\mathcal{O}[\![M]\!]$
$L_1 : A_1 \rightarrow L_2$
$L_2 : A_3 \rightarrow L_3$
$L_3 : A_2 \rightarrow L_4$
$L_4 : A_4 \rightarrow L_5$
$L_5 : A_5 \rightarrow L_6$

In the above example actions A_2 and A_3 are independent, meaning that $\mathbf{A}[\![A_2]\!](\mathbf{A}[\![A_3]\!](\rho, m)) = \mathbf{A}[\![A_3]\!](\mathbf{A}[\![A_2]\!](\rho, m))$ for every $(\rho, m) \in \mathfrak{E} \times \mathfrak{M}$. Considering malware M, we have the trace $\sigma = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$ where:

$$\begin{aligned}
\sigma_1 &= \langle L_1 : A_1 \to L_2, (\rho, m)\rangle = \langle L_1 : A_1 \to L_2, \xi_1^\sigma\rangle \\
\sigma_2 &= \langle L_2 : A_2 \to L_3, (\mathbf{A}[\![A_1]\!](\rho, m))\rangle \\
\sigma_3 &= \langle L_3 : A_3 \to L_4, (\mathbf{A}[\![A_2]\!](\mathbf{A}[\![A_1]\!](\rho, m)))\rangle \\
\sigma_4 &= \langle L_4 : A_4 \to L_5, (\mathbf{A}[\![A_3]\!](\mathbf{A}[\![A_2]\!](\mathbf{A}[\![A_1]\!](\rho, m))))\rangle \\
\sigma_5 &= \langle L_5 : A_5 \to L_6, (\mathbf{A}[\![A_4]\!](\mathbf{A}[\![A_3]\!](\mathbf{A}[\![A_2]\!](\mathbf{A}[\![A_1]\!](\rho, m)))))\rangle \\
&= \langle L_5 : A_5 \to L_6, \xi_5^\sigma\rangle
\end{aligned}$$

while considering the obfuscated version, we have the trace $\delta = \delta_1\delta_2\delta_3\delta_4\delta_5$, where:

$$\begin{aligned}
\delta_1 &= \langle L_1 : A_1 \to L_2, (\rho, m)\rangle = \langle L_1 : A_1 \to L_2, \xi_1^\delta\rangle \\
\delta_2 &= \langle L_2 : A_3 \to L_3, (\mathbf{A}[\![A_1]\!](\rho, m))\rangle \\
\delta_3 &= \langle L_3 : A_2 \to L_4, (\mathbf{A}[\![A_3]\!](\mathbf{A}[\![A_1]\!](\rho, m)))\rangle \\
\delta_4 &= \langle L_4 : A_4 \to L_5, (\mathbf{A}[\![A_2]\!](\mathbf{A}[\![A_3]\!](\mathbf{A}[\![A_1]\!](\rho, m))))\rangle \\
\delta_5 &= \langle L_5 : A_5 \to L_6, (\mathbf{A}[\![A_4]\!](\mathbf{A}[\![A_2]\!](\mathbf{A}[\![A_3]\!](\mathbf{A}[\![A_1]\!](\rho, m)))))\rangle \\
&= \langle L_5 : A_5 \to L_6, \xi_5^\delta\rangle
\end{aligned}$$

Let $Int(M) = \{\sigma_1, \sigma_5\}$. Then $\alpha_{Int(M)}(\alpha_e(\sigma)) = \xi_1^\sigma\xi_5^\sigma$ as well as $\alpha_{Int(M)}(\alpha_e(\delta)) = \xi_1^\delta\xi_5^\delta$, which concludes the example. It is obvious that $\xi_1^\sigma = \xi_1^\delta$, moreover $\xi_5^\delta = \xi_5^\sigma$ follows from the independence of A_2 and A_3.

□

6.5.2 Interesting Behaviors

Program trace semantics expresses malware behaviour on every possible input. It is clear that it may happen that only some of the inputs cause the malware to have a malicious behaviour (think for example of a virus that starts its payload only after a certain date). In this case, maliciousness is properly encoded by a subset of malware traces that identify the so called *interesting behaviours* of the malware. Assume we have an oracle that given a malware M returns the set $T \subseteq \mathbf{S}[\![M]\!]$ of its interesting behaviors. Thus, in order to verify if P is infected by M, we check whether program P matches the set of malicious behaviors T. The following definition characterizes the presence of malware M in a program P in terms of its interesting behaviors T.

Definition 6.19. A program P is infected by a vanilla malware M with interesting behaviors $T \subseteq \mathbf{S}[\![M]\!]$, i.e., $M \hookrightarrow_T P$ if:

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_e(T) \subseteq \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$$

It is interesting to observe that, when program infection is characterized by Definition 6.19, all the results obtained in Section 6.3 still hold if we replace $\mathbf{S}[\![M]\!]$ with T.

Clearly the two abstractions can be composed. In this case, a program P is infected by a malware M if there exists a program restriction that matches the sequences given by the interesting states of the interesting behaviors of the malware, i.e., $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_{Int(M)}(\alpha_e(T)) \subseteq \alpha_{Int(M)}(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$.

6.5.3 Interesting Actions

To conclude, we present a matching relation based on *interesting program actions* rather than environment-memory evolutions. In fact, sometimes, a malicious behavior can be characterized as the execution of a sequence of bad actions. In this case we consider the syntactic information contained in program states. The main difference with purely syntactic approaches is the ability to observe actions in their execution order and not in the order in which they appear in the code. Assume we have an oracle that given a malware M returns the set $Bad \subseteq act[\![M]\!]$ of actions capturing the essence of the malicious behaviour. In this case, in order to verify if program P is infected by malware M, we check whether the execution sequences of bad actions of the malware match the ones of the program.

Definition 6.20. A program P is infected by a vanilla malware M with bad actions Bad, i.e., $M \hookrightarrow_{Bad} P$ if:

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_a(\mathbf{S}[\![M]\!]) \subseteq \alpha_a(\alpha_r(\mathbf{S}[\![P]\!]))$$

Where, given the set $Bad \subseteq act[\![M]\!]$ of bad actions, the abstraction α_a returns the sequence of malicious actions executed by each trace. Formally, given a trace $\sigma = \sigma_1\sigma'$:

$$\alpha_a(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ A_1\alpha_a(\sigma') & \text{if } A_1 \in Bad \\ \alpha_a(\sigma') & \text{otherwise} \end{cases}$$

Even if this abstraction considers syntactic information (program actions), it is able to deal with certain kinds of obfuscations. In fact, considering the sequence of malicious actions in a trace, we observe actions in their execution order, and not in the order in which they are written in the code. This means that, for example, we are able to ignore unconditional jumps and therefore we can deal with code reordering. Once again, abstraction α_a can be combined with interesting states and/or interesting behaviours. For example, program infection can be characterized as the sequences of bad actions present in the

interesting behaviours of malware M, i.e., $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha_a(\alpha_e(T)) \subseteq \alpha_a(\alpha_e(\alpha_r(\mathbf{S}[\![P]\!])))$.

It is clear that the notion of infection given in Definition 6.4 can be weakened in many other ways, following the example given by the above simplifications. This possibility of adjusting malware infection with respect to the knowledge of the malicious behaviour we are searching for proves the flexibility of the proposed semantic framework.

6.6 Relation to Signature Matching

In this section we consider the standard signature matching algorithm for malware detection, and we investigate the effects that it has on program trace semantics in order to to certify the degree of precision of these detection schemes in terms of soundness and completeness properties. We can express the signature of a malware M as a proper subset $S \subseteq M$ of "consecutive" malicious commands, formally $S = C_1, ..., C_n$ where $\forall i \in [1, n-1] : suc[\![C_i]\!] = lab[\![C_{i+1}]\!]$. Given a malware M, $S \subseteq M$ is an *ideal signature* if it unequivocally identifies infection, meaning that $S \subseteq P \Leftrightarrow M \hookrightarrow P$. Signature-based malware detectors, given an ideal signature S of a malware M (provided for example by a perfect oracle OR_S) and a possibly infected program P, syntactically verify infection according to the following test:

$$\textit{Syntactic Test}: \quad S \subseteq P$$

Let us consider the semantic counterpart of the syntactic signature matching test. Given a malware M and its signature $S \subseteq M$, let $lab_s[\![M]\!] = lab[\![S]\!]$ denote the malware restriction identifying the commands composing the signature. Observe that the semantics of the malware restricted to its signature corresponds to the semantics of the signature, i.e., $\alpha_s(\mathbf{S}[\![M]\!]) = \mathbf{S}[\![S]\!]$. Thus, we can say that a program P is infected by a malware M if there exists a restriction of program trace semantics that matches the semantics of the malware restricted to its signature:

$$\textit{Semantic Test}: \quad \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_s(\mathbf{S}[\![M]\!]) = \alpha_r(\mathbf{S}[\![P]\!])$$

which can be equivalently expressed as $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!])$. The following result shows that the syntactic and semantic tests are equivalent, meaning that they detect the same set of infected programs.

Proposition 6.21. *Given a signature S of a malware M we have that:*

$$S \subseteq P \Leftrightarrow \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!])$$

PROOF: ($\Rightarrow$) $S \subseteq P$ means that $\forall C \in S \Rightarrow C \in P$, namely that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : P_r = S$. Therefore, $\alpha_r(\mathbf{S}[\![P]\!]) = \mathbf{S}[\![P_r]\!] = \mathbf{S}[\![S]\!]$. ($\Leftarrow$) If $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!])$, it means that $|\mathbf{S}[\![S]\!]| = |\alpha_r(\mathbf{S}[\![P]\!])|$ and that $\forall \sigma \in \mathbf{S}[\![S]\!]$, $\exists \delta \in \alpha_r(\mathbf{S}[\![P]\!])$ such that $\sigma = \delta = (C_1, \rho_1, m_1), ..., (C_k, \rho_k, m_k)$. This means that for every $\sigma \in \mathbf{S}[\![S]\!]$ and $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$ such that $\sigma = \delta$, we have that $cmd[\![\sigma]\!] = \cup_{i \in [1,k]} C_i = cmd[\![\delta]\!]$, and therefore $S = cmd(\mathbf{S}[\![S]\!]) = cmd(\mathbf{S}[\![P_r]\!]) \subseteq P$, namely $S \subseteq P$.

□

Observe that by applying abstraction α_e to the semantic test we have that $M \hookrightarrow P$ if:

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_e(\alpha_s(\mathbf{S}[\![S]\!])) = \alpha_e(\alpha_r(\mathbf{S}[\![P]\!]))$$

which corresponds to the standard infection condition specified by Definition 6.4 where the semantics of malware M has been restricted to its signature S and the set-inclusion relation has been replaced by equality. It is clear that, in this setting, by replacing $\mathbf{S}[\![M]\!]$ with $\mathbf{S}[\![S]\!]$ we can obtain results analogous to the one proved following Definition 6.4 of infection.

Proving Soundness and Completeness of a Signature-based Detector

Following the proof strategy proposed in Section 6.1.1, we first need to define a trace-based malware detector that is equivalent to the signature-based algorithm, and then we have to prove soundness and completeness for such semantic detector.

Step 1: Designing an equivalent trace-based detector

This point is actually solved by Proposition 6.21. In fact, let $\mathcal{A}_S$ denote the malware detector based on the signature matching algorithm. This syntactic algorithm is based on an oracle OR_S that, given a malware M, returns its ideal signature S such that: $S \subseteq P \Leftrightarrow M \hookrightarrow P$, or, equivalently, $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!]) \Leftrightarrow M \hookrightarrow P$. Let D_S be the trace-based detector that classifies a program P as infected by a malware M with signature S, if $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!])$. From Proposition 6.21 it follows that $\mathcal{A}_S(M, P) = 1$ if and only if $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!])$ if and only if $D_S(M, P) = 1$.

Step 2: Prove soundness and completeness of D_S

Let us identify the class of obfuscating transformations that the trace-based detector D_S is able to handle. The following result shows that D_S is sound if the signature oracle OR_S is perfect, namely D_S is oracle-sound.

Proposition 6.22. *D_S is oracle-sound.*

PROOF: Given a malware M with signature S we have that: $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_r(\mathbf{S}[\![P]\!]) \Rightarrow M \hookrightarrow P$, follows from the hypothesis that OR_S is a perfect oracle that returns an ideal signature.

□

This confirms the general belief that signature matching algorithms have a low false positive rate. In fact, the presence of false positives is caused by the imperfection in the signature extraction process, meaning that in order to improve the signature matching algorithm we have to concentrate in the design of efficient techniques for signature extraction.

Let us introduce the class $\mathbb{O}_S$ of obfuscating transformations that *preserve signatures*. We say that $\mathcal{O}$ preserves signatures, i.e., $\mathcal{O} \in \mathbb{O}_S$, when for every malware M with signature S the semantics of signature S is present in the semantics of the obfuscated malware $\mathcal{O}[\![M]\!]$, formally when:

$$\mathbf{S}[\![S]\!] \subseteq \alpha_s(\mathbf{S}[\![M]\!]) \Rightarrow \exists lab_R[\![\mathcal{O}[\![M]\!]]\!] \in \wp(lab[\![\mathcal{O}[\![M]\!]]\!]) : \mathbf{S}[\![S]\!] \subseteq \alpha_R(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]) \quad (\ddagger)$$

The above condition can equivalently be expressed in syntactic terms as

$$S \subseteq M \Rightarrow S \subseteq \mathcal{O}[\![M]\!]$$

The following result shows that D_S is oracle-complete for $\mathcal{O}$ if and only if $\mathcal{O}$ preserves signatures.

Proposition 6.23. D_S is oracle-complete for $\mathcal{O} \Leftrightarrow \mathcal{O} \in \mathbb{O}_S$.

PROOF: ($\Leftarrow$) Assume that $\mathcal{O} \in \mathbb{O}_S$, then we have to show that: $\mathcal{O}[\![M]\!] \hookrightarrow P \Rightarrow \exists lab_{\mathbf{R}}[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] = \alpha_{\mathbf{R}}(\mathbf{S}[\![P]\!])$. Observe that $\mathcal{O}[\![M]\!] \hookrightarrow P$, means that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : P_r = \mathcal{O}[\![M]\!]$, namely $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_r(\mathbf{S}[\![P]\!]) = \mathbf{S}[\![\mathcal{O}[\![M]\!]]\!]$. From ($\ddagger$), we have that $\exists lab_R[\![\mathcal{O}[\![M]\!]]\!] \in \wp(lab[\![\mathcal{O}[\![M]\!]]\!]) : \mathbf{S}[\![S]\!] = \alpha_R(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!])$, and therefore $\mathbf{S}[\![S]\!] = \alpha_R(\alpha_r(\mathbf{S}[\![P]\!])) = \alpha_{\mathbf{R}}(\mathbf{S}[\![P]\!])$. ($\Rightarrow$) Assume that D_S is complete for $\mathcal{O}$, this means that $\mathcal{O}[\![M]\!] \hookrightarrow P \Rightarrow \exists lab_{\mathbf{R}}[\![P]\!] \in \wp(lab[\![P]\!]) : \mathbf{S}[\![S]\!] \subseteq \alpha_{\mathbf{R}}(\mathbf{S}[\![P]\!])$, meaning that there is a restriction of program P that matches signature S. Thus, program P can be restricted to a signature preserving transformation of M.

□

This means that a signature based detection algorithm $\mathcal{A}_S$ is oracle-complete with respect to the class of obfuscations that preserve malware signatures, namely the ones belonging to $\mathbb{O}_S$. Unfortunately, a lot of commonly used obfuscating transformations do not preserve signatures, namely are not in $\mathbb{O}_S$.

Consider for example the code reordering obfuscation $\mathcal{O}_J$. It is easy to show that $\mathcal{A}_S$ is not complete for $\mathcal{O}_J$. In fact, given a malware M with signature $S \subseteq M$, we have that, in general, $S \not\subseteq \mathcal{O}_J[\![M]\!]$, since jump instructions are inserted between the signature commands changing therefore the signature. In particular, consider signature $S \subseteq M$ such that $S = C_1, ..., C_n$ we have that $S \not\subseteq \mathcal{O}_J[\![M]\!]$, while $S' \subseteq \mathcal{O}_J[\![M]\!]$, where $S' = C'_1 J^* C'_2 J^* ... J^* C'_n$, where J denotes a command implementing an unconditional jump, namely of the form L : `skip` $\rightarrow L'$, and C'_i is given by command C_i with labels updated according to jump insertion. This means that when $\mathcal{O}_J[\![M]\!] \hookrightarrow P$ then $\forall lab_R[\![\mathcal{O}[\![M]\!]]\!] \in \wp(lab[\![\mathcal{O}[\![M]\!]]\!]) : \mathbf{S}[\![S]\!] \not\subseteq \alpha_R(\mathbf{S}[\![\mathcal{O}[\![M]\!]]\!])$. Observe that incompleteness is caused by the fact that D_S, being equivalent to $\mathcal{A}_S$, is strongly related to program syntax, and therefore the insertion of an innocuous jump instruction is able to confuse it.

Following the same strategy, it is possible to show that $\mathcal{A}_S$ is not complete for opaque predicate insertion, semantic NOP insertion and substitution of equivalent commands. Thus, in general, the class of conservative transformations does not preserve malware signatures, i.e., $\mathbb{O}_c \not\subseteq \mathbb{O}_S$, meaning that conservative obfuscations are able to foil signature matching algorithms. Hence, it turns out that $\mathcal{A}_S$ is not complete, namely it is imprecise, for a wide class of obfuscating transformations. This is one of the major drawbacks of signature-based approaches. A common improvement of $\mathcal{A}_S$ consists in considering regular expressions instead of signatures. Namely, given a signature $S = C_1, ..., C_n$, the detector $\mathcal{A}^+_S$ verifies if $C'_1 C^* C'_2 C^* ... C^* C'_n \subseteq P$, where C stands for any command in $\mathbb{C}$ and C'_i is a command with the same action as C_i. It is clear that this allows $\mathcal{A}^+_S$ to deal with the class of obfuscating transformations that are conservative with respect to signatures, as for example code reordering $\mathcal{O}_J$. Let $\mathbb{O}_{cs}$ denote the class of obfuscations that are conservative with respect to signatures, where $\mathcal{O} \in \mathbb{O}_{cs}$ if for every malware M with signature S there exists $S' \subseteq \mathcal{O}[\![M]\!]$ such that $S = C_1 C_2 ... C_n$ and $S' = C'_1 C^* C'_2 C^* ... C^* C'_n$. However, this improvement does not handle all conservative obfuscations in $\mathbb{O}_c$. For example, the substitution of equivalent commands $\mathcal{O}_I$ belongs to $\mathbb{O}_c$ but not to $\mathbb{O}_{cs}$.

6.7 Case Study: Completeness of the Semantics-Aware Malware Detector

In this section we consider an existing detection algorithm and we prove that it is oracle complete for certain obfuscating transformations. Recall that oracle-completeness means that the detection algorithm is complete assuming that the oracles that it uses are perfect. An algorithm called *semantics-aware malware detection* was proposed by Christodorescu, Jha, Seshia, Song, and Bryant [25]. This approach to malware detection uses instruction semantics to identify mali-

Obfuscation	*Completeness of* $\mathcal{A}_{MD}$
Code reordering	Yes
Semantic-nop insertion	Yes
Substitution of equivalent commands	No
Variable renaming	Yes

Table 6.4. Obfuscating transformations considered by $\mathcal{A}_{MD}$

cious behavior in a program, even when obfuscated. The obfuscations considered in [25] are from the set of conservative obfuscations, together with variable renaming. In [25] the authors proved the algorithm to be oracle sound, so we focus in this section on proving its oracle completeness using our abstraction-based framework. The list of obfuscations we consider (shown in Table 6.4) is based on the list described in the semantics-aware malware detection paper.

Description of the Algorithm

The semantics-aware malware detection algorithm $\mathcal{A}_{MD}$ matches a program against a template describing the malicious behavior. If a match is successful, the program exhibits the malicious behavior of the template. Both the template and the program are represented as control flow graphs during the operation of $\mathcal{A}_{MD}$.

The algorithm $\mathcal{A}_{MD}$ attempts to find a subset of program P that matches the commands in malware M, possibly after renaming of variables and locations used in the subset of P. Furthermore, $\mathcal{A}_{MD}$ checks that any def-use relationship that holds in the malware also holds in the program, across program paths that connect consecutive commands in the subset.

A control flow graph $G = (V, E)$ is a graph with the vertex set V representing program commands, and edge set E representing control-flow transitions from one command to its successor(s). For our language the control-flow graph (CFG) can be easily constructed as follows:

- For each command $C \in \mathbb{C}$, create a CFG node annotated with that command, $v_{lab[\![C]\!]}$. Correspondingly, we write $C[\![v]\!]$ to denote the command at CFG node v.
- For each command $C = L_1 : A \rightarrow S$, where $S \in \wp(\mathbb{L})$, and for each label $L_2 \in S$, create a CFG edge (v_{L_1}, v_{L_2}).

Consider a path θ through the CFG from node v_1 to node v_k, $\theta = v_1 \rightarrow \ldots \rightarrow v_k$. There is a corresponding sequence of commands in the program P, written $P|_\theta = \{C_1, \ldots, C_k\}$, where $C_i = C[\![v_i]\!]$. Then we can express the set of states possible after executing the sequence of commands $P|_\theta$ as $\mathbf{C}^k[\![P|_\theta]\!](C_1, \rho, m)$, by extending the transition relation $\mathbf{C}$ to a set of states, such that $\mathbf{C} : \wp(\Sigma) \rightarrow \wp(\Sigma)$. Let us define the following basic functions:

$$mem[\![(C,\rho,m)]\!] = m \qquad env[\![(C,\rho,m))]\!] = \rho$$

The algorithm takes as inputs the *CFG* for the template, $G^T = (V^T, E^T)$, and the binary file for the program, $File[\![P]\!]$. For each path θ in G^T, the algorithm proceeds in two steps:

1. Identify a one-to-one map from template nodes in the path θ to program nodes, denoted by $\mu_\theta : V^T \to V^P$. A template node n^T can match a program node n^P if the top-level operators in their actions are identical. This map induces a map $\nu_\theta : \mathbb{X}^T \times V^T \to \mathbb{X}^P$ from variables at a template node to variables at the corresponding program node, such that when renaming the variables in the template command $C[\![n^T]\!]$ according to the map ν_θ, we obtain the program command $C[\![n^P]\!] = C[\![n^T]\!][X/\nu_\theta\left(X, n^T\right)]$.
   This step makes use of the *CFG* oracle OR_{CFG} that returns the control-flow graph $G^P = (V^P, E^P)$ of a program P, given P's binary-file representation $File[\![P]\!]$.
2. Check whether the program preserves the def-use dependencies that are true on the template path θ. For each pair of template nodes m^T, n^T on the path θ, and for each template variable X^T defined in $act[\![C[\![m^T]\!]]\!]$ and used in $act[\![C[\![n^T]\!]]\!]$, let λ be a program path $\mu_\theta(v_1^T) \to \ldots \to \mu_\theta(v_k^T)$, where $m^T \to v_1^T \to \ldots \to v_k^T \to n^T$ is part of the path θ in the template *CFG*. λ is therefore a program path connecting the program *CFG* node corresponding to m^T with the program *CFG* node corresponding to n^T. We denote by $T|_\theta = \left\{C[\![m^T]\!], C[\![v_1^T]\!], \ldots, C[\![v_k^T]\!], C[\![n^T]\!]\right\}$ the sequence of commands corresponding to the template path θ.
   The def-use preservation check can be expressed formally as follows

$$\forall \rho \in \mathfrak{E}, \forall m \in \mathfrak{M}, \forall s \in \mathbf{C}^k[\![P|_\lambda]\!]\left(\mu_\theta(v_1), \rho, m\right) : \\ \mathbf{E}[\![\nu_\theta(X^T, v_1)]\!](\rho, m) = \mathbf{E}[\![\nu_\theta(X^T, n^T)]\!]\left(env[\![s]\!], mem[\![s]\!]\right)$$

   The above formula checks weather the value of the program variable corresponding to the template variable X^T before the execution of λ remains constant during the execution of λ. This check is implemented in $\mathcal{A}_{MD}$ as a query to a *semantic-nop oracle* OR_{SNop}. The semantic-nop oracle determines whether the value of a variable X before the execution of a code sequence $\psi \subseteq P$ is equal to the value of a variable Y after the execution of ψ.

The semantics-aware malware detector $\mathcal{A}_{MD}$ makes use of two oracles, OR_{CFG} and OR_{SNop}, described in Table 6.5. Thus $\mathcal{A}_{MD} = D^{\mathcal{OR}}$, for the set of oracles $\mathcal{OR} = \{OR_{CFG}, OR_{SNop}\}$. Our goal is then to verify whether $\mathcal{A}_{MD}$ is oracle complete with respect to the obfuscations from Table 6.4.

We follow the proof strategy proposed in Section 6.1.1. First, in step 1 below, we develop a trace-based detector D_{Tr} based on an abstraction α, and show that

Oracle	*Notation*
CFG oracle	$OR_{CFG}\,(File[\![P]\!])$ Returns the control-flow graph of the program P, given its binary-file representation $File[\![P]\!]$.
Semantic-nop oracle	$OR_{SNop}(\psi, X, Y)$ Determines whether the value of variable X before the execution of code sequence $\psi \subseteq P$ is equal to the value of variable Y after the execution of ψ.

Table 6.5. Oracles used by $\mathcal{A}_{MD}$.

$D^{\mathcal{OR}} = \mathcal{A}_{MD}$ and D_{Tr} are equivalent. This equivalence of detectors holds only if the oracles in $\mathcal{OR}$ are perfect. Then, in step 2, we show that D_{Tr} is complete with respect to the obfuscations of interest.

Step 1: Design an Equivalent Trace-Based Detector

We can model the algorithm for semantics-aware malware detection using two abstractions, α_{SAMD} and α_{Act}. The abstraction α that characterizes the trace-based detector D_{Tr} is given by the composition of these two abstractions, i.e., $\alpha = \alpha_{Act} \circ \alpha_{SAMD}$. We will show that D_{Tr} is equivalent to $\mathcal{A}_{MD} = D^{\mathcal{OR}}$, when the oracles in $\mathcal{OR}$ are perfect.

The abstraction α_{SAMD}, when applied to a trace $\sigma \in \mathbf{S}[\![P]\!]$, with $\sigma = (C'_1, \rho'_1, m'_1) \ldots (C'_n, \rho'_n, m'_n)$, to a set of variable maps $\{\pi_i\}$, and a set of location maps $\{\gamma_i\}$, returns an abstract trace:

$$\alpha_{SAMD}(\sigma, \{\pi_i\}, \{\gamma_i\}) = (C_1, \rho_1, m_1) \ldots (C_n, \rho_n, m_n)$$
$$\text{if } \forall i,\ 1 \le i \le n\ : act[\![C_i]\!] = act[\![C'_i]\!][X/\pi_i(X)], lab[\![C_i]\!] = \gamma_i(lab[\![C'_i]\!]),$$
$$suc[\![C_i]\!] = \gamma_i(suc[\![C'_i]\!]), \rho_i = \rho'_i \circ \pi_i^{-1}, m_i = m'_i \circ \gamma_i^{-1}$$

Otherwise, if the condition does not hold, $\alpha_{SAMD}(\sigma, \{\pi_i\}, \{\gamma_i\}) = \epsilon$. A map $\pi_i : var[\![P]\!] \to \mathbb{X}$ renames program variables such that they match malware variables, while a map $\gamma_i : lab[\![P]\!] \to \mathbb{L}$ reassigns program memory locations to match malware memory locations.

The abstraction α_{Act} simply strips all labels from the commands in a trace $\sigma = (C_1, \rho_1, m_1)\sigma'$, as follows:

$$\alpha_{Act}(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ (act[\![C_1]\!], \rho_1, m_1)\alpha_{Act}(\sigma') & \text{otherwise} \end{cases}$$

Definition 6.24. An *α-semantic malware detector* is a malware detector on the abstraction α, i.e., it classifies the program P as infected by a malware M, $M \hookrightarrow P$, if

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha(\mathbf{S}[\![M]\!]) \subseteq \alpha(\alpha_r(\mathbf{S}[\![P]\!]))$$

By this definition, a semantic malware detector (from Definition 6.4) is a special instance of the α-semantic malware detector, for $\alpha = \alpha_e$. Let D_{Tr} be a $\alpha_{Act} \circ \alpha_{SAMD}$-semantic malware detector. The following result shows that D_{Tr} is equivalent to the semantics-aware malware detector $\mathcal{A}_{MD}$. In particular, the proof has two parts, to show that $\mathcal{A}_{MD}(P, M) = 1 \Rightarrow D_{Tr}(\mathbf{S}[\![P]\!], \mathbf{S}[\![M]\!]) = 1$, and then to show the reverse. For the first implication, it is sufficient to show that for any path θ in the *CFG* of M and the path χ in the *CFG* of P, such that θ and χ are found as related by the algorithm $\mathcal{A}_{MD}$, the corresponding traces are equal when abstracted by $\alpha_{Act} \circ \alpha_{SAMD}$. The proof for the second implication proceeds by showing that any two traces $\sigma \in \mathbf{S}[\![M]\!]$ and $\delta \in \mathbf{S}[\![P]\!]$, that are equal when abstracted by $\alpha_{Act} \circ \alpha_{SAMD}$, have corresponding paths through the *CFG*s of M and P, respectively, such that these paths satisfy the conditions in the algorithm $\mathcal{A}_{MD}$. Both parts of the proof depend on the oracles used by $\mathcal{A}_{MD}$ to be perfect.

Proposition 6.25. The semantics-aware malware detector algorithm $\mathcal{A}_{MD}$ is equivalent to the $\alpha_{Act} \circ \alpha_{SAMD}$-semantic malware detector D_{Tr}. In other words, $\forall P, M \in \mathbb{P}$, we have that $\mathcal{A}_{MD}(P, M) = D_{Tr}(\mathbf{S}[\![P]\!], \mathbf{S}[\![M]\!])$.

PROOF: To show that $\mathcal{A}_{MD} = D_{Tr}$, we can equivalently show that $\forall P, M \in \mathbb{P}$: $\mathcal{A}_{MD}(P, M) = 1 \iff \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$, $\exists\{\pi_i\}_{i\geq 1}$, and $\exists\{\gamma_i\}_{i\geq 1}$ such that $\alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathbf{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\}))$. Since π_i renames variables only from P (i.e., $\forall V \in \mathbb{V} \setminus var[\![P]\!]$, π_i is the identity function, $\pi_i(X) = X$), and similarly γ_i remaps locations only from P, then we have that $\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\}) = \mathbf{S}[\![M]\!]$.

($\Rightarrow$) Assume that $\mathcal{A}_{MD}(P, M) = 1$. Let G^M be the *CFG* of malware M and let $Path(G^M)$ denote the set of all paths on G^M. We can construct the restriction $lab_r[\![P]\!]$ from the path-sensitive map μ_θ as follows:

$$lab_r[\![P]\!] = \bigcup_{\theta \in Paths(G^M)} \left\{ \; lab[\![C[\![\mu_\theta\left(v^M\right)]\!]]\!] \;\middle|\; v^M \in \theta \; \right\}$$

Following the above construction $lab_r[\![P]\!]$ collects the labels of program commands whose nodes corresponds to a template node through μ_θ. The variable maps $\{\pi_i\}$ can be defined based on ν_θ. For a path $\theta = v_1^M \rightarrow \ldots \rightarrow v_k^M$, $\pi_i(X) = \nu_\theta\left(X, v_i^M\right)$. Similarly, $\gamma_i(L) = L'$ if $lab[\![C[\![v_i^M]\!]]\!] = L'$ and $lab[\![C[\![\mu_\theta\left(v_i^M\right)]\!]]\!] = L$.

Let $\sigma \in \mathbf{S}[\![M]\!]$ and denote by $\theta = v_1^M \rightarrow \ldots \rightarrow v_k^M$ the *CFG* path corresponding to this trace. By algorithm $\mathcal{A}_{MD}$, there exists a path χ in the *CFG* of P of the form:

$$\ldots \rightarrow \mu_\theta\left(v_1^M\right) \rightarrow \ldots \rightarrow \mu_\theta\left(v_k^M\right) \rightarrow \ldots$$

Let $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$ be the trace corresponding to the path χ in G^P,

$$\delta = \ldots \langle C[\![\mu_\theta\left(v_1^M\right)]\!], \rho_1^P, m_1^P\rangle \ldots \langle C[\![\mu_\theta\left(v_k^M\right)]\!], \rho_k^P, m_k^P\rangle \ldots$$

For two states i and $j > i$ of the trace σ, denote the intermediate states in the trace δ by $\langle C_1'^P, \rho_1'^P, m_1'^P\rangle \dots \langle C_l'^P, \rho_l'^P, m_l'^P\rangle$, i.e., $\delta =$

$$..\langle C[\![\mu_\theta\left(v_i^M\right)]\!], \rho_i^P, m_i^P\rangle\langle C_1'^P, \rho_1'^P, m_1'^P\rangle \dots \langle C_l'^P, \rho_l'^P, m_l'^P\rangle\langle C[\![\mu_\theta\left(v_j^M\right)]\!], \rho_j^P, m_j^P\rangle..$$

From step 1 of algorithm $\mathcal{A}_{MD}$, we have that the following holds:

$$\begin{aligned} act[\![C[\![\mu_\theta\left(v_i^M\right)]\!]]\!][X/\pi_i(X)] &= act[\![C[\![v_i^M]\!]]\!] \\ \gamma_i\left(lab[\![C[\![\mu_\theta\left(v_i^M\right)]\!]]\!]\right) &= lab[\![C[\![v_i^M]\!]]\!] \\ \gamma_i\left(suc[\![C[\![\mu_\theta\left(v_i^M\right)]\!]]\!]\right) &= suc[\![C[\![v_i^M]\!]]\!] \end{aligned}$$

From step 2 of algorithm $\mathcal{A}_{MD}$, we know that for any template variable X^M that is defined in $C[\![v_i^M]\!]$ and used in $C[\![v_j^M]\!]$ (for $1 \leq i < j \leq k$), we have that:

$$\mathbf{E}[\![\nu_\theta(X^M, v_i^M)]\!](\rho, m) = \mathbf{E}[\![\nu_\theta(X^M, v_j^M)]\!](env[\![s]\!], mem[\![s]\!])$$

where $s \in \mathbf{C}^l\big(\langle\mu\left(v_i^M\right)\rangle, \rho, m\big)$. Since we have that $act[\![C[\![\mu_\theta\left(v_i^M\right)]\!]]\!][X/\pi_i(X)] = act[\![C[\![v_i^M]\!]]\!]$, it follows that $\rho_i^P(\nu_\theta(X^M, v_i^M)) = \rho_j^P(\nu_\theta(X^M, v_j^M))$. Moreover, since $\rho_i^M(X^M) = \rho_j^M(X^M)$, then we can write $\rho_i^M = \rho_i^P \circ \pi_i$. Similarly, $m_i^M = m_i^P \circ \gamma_i$. Then it follows that for every $\sigma \in \mathbf{S}[\![M]\!]$, there exists $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$ such that:

$$\begin{aligned} \alpha_{Act}(\alpha_{SAMD}(\sigma, \{\pi_i\}, \{\gamma_i\})) &= \alpha_{Act}(\sigma) \\ &= \alpha_{Act}(\alpha_{SAMD}(\delta, \{\pi_i\}, \{\gamma_i\})) \end{aligned}$$

Thus, $\alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathbf{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\}))$.

($\Leftarrow$) Assume that $lab_r[\![P]\!]$, $\{\pi_i\}_{i\geq 1}$, and $\{\gamma_i\}_{i\geq 1}$ exist such that:

$$\alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathbf{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\}))$$

We will show that $\mathcal{A}_{MD}$ returns 1, that is, the two steps of the algorithm complete successfully.

Let $\sigma \in \alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\}))$, with

$$\sigma = \langle A_1, \rho_1^M, m_1^M\rangle \dots \langle A_k, \rho_k^M, m_k^M\rangle.$$

Then there exists $\sigma' \in \mathbf{S}[\![M]\!]$

$$\sigma' = \langle C_1^M, \rho_1^M, m_1^M\rangle \dots \langle C_k^M, \rho_k^M, m_k^M\rangle,$$

such that $\forall i$, $act[\![C_i^M]\!][X/\pi_i(X)] = A_i$. Similarly, there exists $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$, with $\delta = \langle C_1^P, \rho_1^P, m_1^P\rangle \dots \langle C_k^P, \rho_k^P, m_k^P\rangle$, such that $\forall i$, $act[\![C_i^P]\!][X/\pi_i(X)] = A_i$, $\rho_i^P = \rho_i^M \circ \pi_i^{-1}$, and $m_i^P = m_i^M \circ \gamma_i^{-1}$. In other words, we have that

$\sigma = \alpha_{Act}(\alpha_{SAMD}(\sigma', \{\pi_i\}, \{\gamma_i\})) = \alpha_{Act}(\alpha_{SAMD}(\delta', \{\pi_i\}, \{\gamma_i\}))$, where σ' is a malware trace and δ' is a trace of the restricted program P_r induced by $lab_r[\![P]\!]$. For each pair of traces (σ, δ) chosen as above, we can define a map μ from nodes in the *CFG* of M to nodes in the *CFG* of P by setting $\mu(v_{lab[\![C_i^M]\!]}) = v_{lab[\![C_i^P]\!]}$. Without loss of generality, we assume that $lab[\![M]\!] \cap lab[\![P]\!] = \emptyset$. Then μ is a one-to-one, onto map, and step 1 of algorithm $\mathcal{A}_{MD}$ is complete.

Consider a variable $X^M \in var[\![M]\!]$ that is defined by action A_i and later used by action A_j in the trace σ', for $j > i$, such that $\rho^M_{i+1}(X^M) = \rho^M_j(X^M)$. Let X^P_i be the program variable corresponding to X^M at program command C^P_i, and X^P_j the program variable corresponding to X^M at program command C^P_j:

$$x^P_i = \nu(X^M, v_{lab[\![C_i^M]\!]}) \qquad\qquad x^P_j = \nu(X^M, v_{lab[\![C_j^M]\!]})$$

If $\delta \in \alpha_r(\mathbf{S}[\![P]\!])$, then there exists a $\delta' \in \mathbf{S}[\![P]\!]$ of the form:

$$\delta' = \ldots \langle C^P_i, \rho^P_i, m^P_i \rangle \ldots \langle C^P_j, \rho^P_j, m^P_j \rangle \ldots$$

where $1 \leq i < j \leq k$. Let θ be a path in the *CFG* of P, $\theta = v^P_1 \rightarrow \ldots \rightarrow v^P_k$, such that $v^P_{lab[\![C_i^P]\!]} \rightarrow v^P_1 \rightarrow \ldots \rightarrow v^P_k \rightarrow v^P_{lab[\![C_j^P]\!]}$ is also a path in the *CFG* of P. Since $\rho^M_{i+1}(X^M) = \rho^M_j(X^M)$, then $\rho^P_{suc[\![C_i^P]\!]}(X^P_i) = \rho^M_{i+1}(\pi_i(X^P_i)) = \rho^M_{i+1}(X^M) = \rho^M_j(X^M) = \rho^P_j(\pi_j(X^P_j)) = \rho^P_j(X^P_j)$. But $suc[\![C^P_i]\!] = lab[\![C^P[\![v_1]\!]]\!]$ in the trace δ'. As $\mathbf{E}[\![X^P_i]\!](\rho, m) = \rho(x^P_i)$, it follows that

$$\mathbf{E}[\![\nu(X^M, v_{lab[\![C_i^M]\!]})]\!](\rho, m) = \mathbf{E}[\![\nu(X^M, v_{lab[\![C_j^M]\!]})]\!]\,(env[\![s]\!], mem[\![s]\!])$$

for any $\rho \in \mathfrak{E}$, any $m \in \mathfrak{M}$, and any state s of P at the end of executing the path θ, i.e., $s \in \mathbf{C}^k[\![P|_\theta]\!](\langle \mu(v^P_{lab[\![C_i^P]\!]}), \rho, m\rangle)$. If the semantic-nop oracle queried by $\mathcal{A}_{MD}$ is complete, then the second step of the algorithm is successful. Thus $\mathcal{A}_{MD}(P, M) = 1$.

□

Now we can characterize the semantics-aware malware detector algorithm $\mathcal{A}_{MD}$ as the following infection condition on program trace semantics.

Definition 6.26. A program P is infected by a vanilla malware M, i.e., $M \hookrightarrow P$, if:

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]),\ \{\pi_i\}_{i\geq 1},\ \{\gamma_i\}_{i\geq 1}\ :$$
$$\alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathbf{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\})).$$

Step 2: Prove Completeness of the Trace-Based Detector

We are interested in finding out which classes of obfuscations are handled by the semantics-aware malware detector $\mathcal{A}_{MD}$. We check the validity of the completeness condition expressed in Definition 6.6. In other words, if the program is infected with an obfuscated variant of the malware, then the semantics-aware detector should return 1. Consider for example the code reordering obfuscation that inserts `skip` commands into the program and changes the labels of existing commands. In this case, the restriction α_r "eliminates" the inserted `skip` commands, while the α_{Act} abstraction allows for trace comparison while ignoring command labels. Thus, the detector D_{Tr} is oracle-complete with respect to the code-reordering obfuscation.

Proposition 6.27. The semantics-aware malware detector $\mathcal{A}_{MD}$ is oracle-complete with respect to the code-reordering obfuscation $\mathcal{O}_J$:

$$\mathcal{O}_J[\![M]\!] \hookrightarrow P \Rightarrow \begin{cases} \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!],\ \{\pi_i\}_{i\geq 1},\ \{\gamma_i\}_{i\geq 1} : \\ \quad \alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \\ \quad\quad \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathbf{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\})) \end{cases}$$

PROOF: If $\mathcal{O}_J[\![M]\!] \hookrightarrow P$, and given that $\mathcal{O}_J$ inserts only `skip` commands into a program, then $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $P_r = \mathcal{O}_J[\![M]\!] \setminus \mathit{Skip}$, where *Skip* is a set of `skip` commands inserted by $\mathcal{O}_J$, as defined in Section 6.4. Let $M' = \mathcal{O}_J(M) \setminus \mathit{Skip}$. Then $\alpha_r(\mathbf{S}[\![P]\!]) = \mathbf{S}[\![M']\!]$. Thus we have to prove that

$$\alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M']\!], \{\pi_i\}, \{\gamma_i\}))$$

for some $\{\pi_i\}$ and $\{\gamma_i\}$. As $\mathcal{O}_J[\![M]\!]$ does not rename variables or change memory locations, we can set π_i and γ_j, for all i and j, to be the respective identity maps, $\pi_i = Id_{var[\![P]\!]}$ and $\gamma_j = Id_{lab[\![P]\!]}$. From this observation, it follows that $\alpha_{SAMD}(\mathbf{S}[\![M']\!], \{Id_{var[\![P]\!]}\}, \{Id_{lab[\![P]\!]}\}) = \mathbf{S}[\![M']\!]$ and $\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{Id_{var[\![P]\!]}\}, \{Id_{lab[\![P]\!]}\}) = \mathbf{S}[\![M]\!]$. Thus, it remains to show that $\alpha_{Act}(\mathbf{S}[\![M]\!]) \subseteq \alpha_{Act}(\mathbf{S}[\![M']\!])$. By the definition of $\mathcal{O}_J$, we have that $M' = \mathcal{O}_J[\![M]\!] \setminus \mathit{Skip} = (M \setminus S) \cup \eta(S)$, for some $S \subset M$. But $\eta(S)$ only updates the labels of the commands in S, and thus we have:

$$\begin{aligned} \alpha_{Act}(\mathbf{S}[\![M']\!]) &= \alpha_{Act}(\mathbf{S}[\![(M \setminus S) \cup \eta(S)]\!]) \\ &= \alpha_{Act}(\mathbf{S}[\![M]\!]). \end{aligned}$$

It follows that $\alpha_{Act}(\mathbf{S}[\![M]\!]) \subseteq \alpha_{Act}(\mathbf{S}[\![\mathcal{O}_J[\![M]\!] \setminus \mathit{Skip}]\!])$.

□

Similar proofs confirm that D_{Tr} is oracle-complete with respect to variable renaming and semantic NOP insertion.

Proposition 6.28. The semantics-aware malware detector $\mathcal{A}_{MD}$ is oracle-complete with respect to the variable-renaming obfuscation $\mathcal{O}_v$.

Proposition 6.29. The semantics-aware malware detector $\mathcal{A}_{MD}$ is oracle-complete with respect to the semantic NOP insertion obfuscation $\mathcal{O}_N$.

Additionally, D_{Tr} is oracle-complete with respect to a limited version of substitution of equivalent commands, when the commands in the original malware M are not substituted with equivalent commands. Unfortunately, D_{Tr} is not oracle-complete with respect to all conservative obfuscations, as the following result illustrates.

Proposition 6.30. The semantics-aware malware detector $\mathcal{A}_{MD}$ is not oracle-complete with respect to all conservative obfuscations $\mathcal{O}_c \in \mathbb{O}_c$.

PROOF: To prove that semantics-aware malware detection is not complete on α_{SAMD} w.r.t. all conservative obfuscations, it is sufficient to find one conservative obfuscation such that

$$\alpha_{Act}(\alpha_{SAMD}(\mathbf{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathbf{S}[\![\mathcal{O}_c(M)]\!]), \{\pi_i\}, \{\gamma_i\})) \quad (6.3)$$

cannot hold for any restriction $lab_r[\![\mathcal{O}_c[\![M]\!]]\!] \in \wp(lab[\![\mathcal{O}_c[\![M]\!]]\!])$ and any maps $\{\pi_i\}_{i\geq 1}$ and $\{\gamma_i\}_{i\geq 1}$.

Consider an instance of the substitution of equivalent commands obfuscating transformation $\mathcal{O}_I$ that substitutes the action of at least one command for each path through the program (i.e., $\mathbf{S}[\![P]\!] \cap \mathbf{S}[\![\mathcal{O}_I[\![P]\!]]\!] = \emptyset$) – for example, the transformation could modify the command at the start label of the program. Assume that $\exists\{\pi_i\}_{i\geq 1}$ and $\exists\{\gamma_i\}_{i\geq 1}$ such that Equation 6.3 holds, where $\mathcal{O}_c = \mathcal{O}_I$. Then $\exists\sigma \in \mathbf{S}[\![M]\!]$ and $\exists\delta \in \mathbf{S}[\![\mathcal{O}_I[\![M]\!]]\!]$ such that $\alpha_{Act}(\sigma) = \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\delta), \{\pi_i\}, \{\gamma_i\}))$. As $|\sigma| = |\delta|$, we have that $\alpha_r(\delta) = \delta$. If $\sigma = \ldots\langle C_i, \rho_i, m_i\rangle\ldots$ and $\delta = \ldots\langle C'_i, \rho'_i, m'_i\rangle\ldots$, then we have that $\forall i, act[\![C_i]\!] = act[\![C'_i]\!][X/\pi_i(X)]$. But from the definition of the obfuscating transformation $\mathcal{O}_I$ above, we know that $\forall\sigma \in \mathbf{S}[\![M]\!]$, $\forall\delta \in \mathbf{S}[\![\mathcal{O}_I[\![M]\!]]\!]$, $\exists i \geq 1$ such that $C_i \in \sigma$, $C'_i \in \delta$, and $\forall\pi : \mathbb{X} \to \mathbb{X}, act[\![C_i]\!] \neq act[\![C'_i]\!][X/\pi(X)]$. Hence we have a contradiction.

□

The cause for this incompleteness is the fact that the abstraction applied by D_{Tr} still preserves some of the actions from the program. Consider an instance of the substitution of equivalent commands obfuscating transformation $\mathcal{O}_I$ that substitutes the action of at least one command for each path through the malware (i.e., $\mathbf{S}[\![M]\!] \cap \mathbf{S}[\![\mathcal{O}_I[\![M]\!]]\!] = \emptyset$). For example, the transformation could modify the command at M's start label. Such an obfuscation, because it affects at least one action of M on every path through the program $P = \mathcal{O}_I[\![M]\!]$, will defeat the detector.

6.8 Discussion

Malware detectors have traditionally relied upon syntactic approaches, typically based on signature-matching algorithms. While such approaches are simple, they are easily defeated by obfuscations. To address this problem, we present a semantics-based framework within which one can specify what it means for a malware detector to be sound and complete, and reason about the completeness and soundness of malware detectors with respect to various classes of obfuscations. For example, in this framework, it is possible to show that the signature-based malware detector is generally sound but not complete, as well as that the semantics-aware malware detector proposed by Christodorescu et al. is complete with respect to some commonly used malware obfuscations.

Our framework uses a trace semantics to characterize the behaviors of both the malware and the program being analyzed. It shows how we can get around the effects of obfuscations by using abstract interpretation to "hide" irrelevant aspects of these behaviors. Thus, given an obfuscating transformation $\mathcal{O}$, the key point is to characterize the proper semantic abstraction that recognises infection even if the malware is obfuscated through $\mathcal{O}$. So far, given an obfuscating transformation $\mathcal{O}$, we assume that the proper abstraction α, which discards the details changed by the obfuscation and preserves maliciousness, is provided by the malware detector designer. We are currently investigating how to design a systematic (ideally automatic) methodology for deriving an abstraction α that leads to a sound and complete semantic malware detector. As a first step in this direction, we observe that if abstraction α is preserved by the obfuscation $\mathcal{O}$ then the malware detection is complete, i.e., no false negatives. However, preservation is not enough to eliminate false positives. Hence, an interesting research task consists in characterizing the set of semantic abstractions that prevents false positives. This, characterization may help us in the design of suitable abstractions that are able to deal with a given obfuscation.

Other approaches to the automatic design of abstraction α can rely on *monitoring* malware execution in order to extract its malicious behaviours, i.e., the set of malicious (abstract) traces that characterizes the malign intent. The idea is that every time that a malware exhibits a malicious intent (for example every time it violates some security policies) the behaviour is added to the set of malicious ones. Another possibility we are interested in is the use of *data mining* techniques to extract maliciousness in malware behaviours. In this case, given a sufficient wide class of malicious variants we can analyze their semantics and use data mining to extract common features.

For future work in designing malware detectors, an area of great promise is that of detectors that focus on interesting actions. Depending on the execution environment, certain states are reachable only through particular actions. For example, system calls are the only way for a program to interact with OS-mediated

resources such as files and network connections. If the malware is characterized by actions that lead to program states in an unique, unambiguous way, then all applicable obfuscation transformations are conservative. As we showed, a semantic malware detector that is both sound and complete for a class of conservative obfuscations exists, if an appropriate abstraction can be designed. In practice, such an abstraction cannot be precisely computed, due to undecidability of program trace semantics – a future research task is to find suitable approximations that minimize false positives while preserving completeness.

One further step would be to investigate whether and how model checking techniques can be applied to detect malware. Some works along this line already exist [84]. Observe that abstraction α actually defines a set of program traces that are equivalent up to $\mathcal{O}$. In model checking, sets of program traces are represented by formulae of some linear/branching temporal logic. Hence, we aim at defining a temporal logic whose formulae are able to express normal forms of obfuscations together with operators for composing them. This would allow us to use standard model checking algorithms to detect malware in programs. This could be a possible direction to follow in order to develop a practical tool for malware detection based on our semantic model. We expect this semantics-based tool to be significantly more precise than existing virus scanners.

7

Conclusions

In this dissertation we consider code obfuscation as a defense technique for preventing attacks to the intellectual property of programs, as well as a malicious transformation used by malware writers to foil misuse detection. In order to contrast some well known drawbacks of both scenarios, such as the lack of rigorous theoretical bases for software protection and the purely syntactic basis of misuse detection, we have proposed a formal approach to code obfuscation and malware detection based on program semantics and abstract interpretation.

Recently, it has been shown how programs can be seen as abstractions of their semantics and how syntactic transformations can be specified as approximations of their semantic counterpart [44]. In particular, this result shows that abstract interpretation provides the right setting in which to formalize the relationship between code obfuscation and its effects on program semantics. We propose a semantic framework which relies on a semantics-based definition of code obfuscation and on an abstract interpretation-based model for attackers. In fact, we characterize the obfuscating behaviour of a program transformation $\mathbb{t}$ in terms of the most concrete semantic property $\delta_{\mathbb{t}}$ it preserves. Given a transformation $\mathbb{t}$, property $\delta_{\mathbb{t}}$ precisely expresses the amount of information still available after the obfuscation $\mathbb{t}$, namely what the obfuscated program might reveal to attackers about the original program. Following our definition, any program transformation $\mathbb{t}$ can be seen as an obfuscator defeating the attackers that are interested in something more precise than $\delta_{\mathbb{t}}$. This is one of the reason why our definition turns out to be a generalization of the standard notion of code obfuscation, which requires obfuscating transformations to preserve denotational program semantics [34]. In this formal setting, it comes natural to model attackers as semantic properties, namely as abstractions of trace semantics, where the abstraction modeling the attacker precisely encodes the semantic properties in which the attacker is interested. Hence, obfuscations as well as attackers are characterized as semantic properties, meaning that they can be compared and related to each other in the lattice of abstract interpretation. In fact, given a

program transformation it is possible to define the class of attackers it defeats, and given an attacker we can identify the family of obfuscations it is able to break. Investigating the semantics aspects of code obfuscation is crucial in order to understand the true potency of these transformations. Following our definition, code obfuscation aims at confusing program syntax while preserving an approximation of its semantics. Thus, being able to precisely identify what can be deduced of the original program behaviour when observing an obfuscated version of it, tells us the maximal amount of information that an attacker can recover from the obfuscated program and therefore if a given defense technique is appropriate in a certain scenario.

We show our semantic framework in action by investigating the effects that control code obfuscation through opaque predicate insertion has on program trace semantics. We define a semantic transformation t^{OP} that transforms sets of traces, namely program semantics, according to opaque predicate insertion. Next we show how an iterative algorithm for opaque predicate insertion can be derived from t^{OP}, following the methodology proposed in [44]. It is clear that, in order to recover a program from opaque predicate insertion, an attacker has to identify the predicates that are opaque and eliminate them together with their never executed branches. For this reason, we say that an attacker breaks an opaque predicate when it is able to detect its opaqueness. It turns out that, modeling, as usual, attackers as abstract domains, the ability of an attacker to break certain classes of opaque predicates can be expressed as a completeness problem in the abstract interpretation sense. In particular, our completeness result holds for two interesting classes of numerical opaque predicates commonly used by existing obfuscating tools. The importance of this result relies in the fact that there exists a systematic way for minimally refining an abstract domain in order to make it complete for a given function. This means that, given an attacker A and an opaque predicate P^T it is always possible to formalize the amount of information needed by A in order to break P^T. It is clear that the bigger the amount of information needed by A, the greater is the degree of protection provided by opaque predicate P^T. Obviously, this can be used to compare the efficiency of an opaque predicate in contrasting different attackers, as well as the resilience of different opaque predicates against a certain attack.

A recent result by Christodorescu et al. [25] confirms the potential benefits of a semantics-based approach to malware detection. Following this observation, and the work already done on the semantic aspects of code obfuscation, we address the malware detection problem from a semantic point of view. The basic idea of our approach is to model both program and malware behaviours through their trace semantics, and to use abstract interpretation to hide irrelevant aspects of these behaviours. Given an obfuscating transformation $\mathcal{O}$, our idea is to identify a suitable abstraction α that is able to discards the details changed by the obfuscation while preserving maliciousness. Thus, checking if the semantics

of a program P matches the semantics of a malware M up to abstraction α we are able to decide if program P is infected with a variant of malware M obtained through obfuscation $\mathcal{O}$. Obviously the key point of this approach is the design of a suitable abstraction α able to deal with as many obfuscating transformations as possible. In order to determine a common pattern for the design of a useful abstraction we have analyzed the effects of different obfuscating transformations on program trace semantics. We provide a classification of obfuscating transformations based on such semantic effects. In particular, an obfuscation $\mathcal{O}$ is conservative when for each trace σ of the original program semantics there exists a trace δ in the semantics of the transformed program such that σ is a subtrace of δ, namely such that all the states of σ are present in δ in the same order. When $\mathcal{O}$ does not satisfy this condition the transformation is non-conservative. We prove that most obfuscating transformation typically used by malware to avoid detection are conservative, and that the property of being conservative is preserved by composition. Moreover, for the widely used class of conservative obfuscations we are able to provide a suitable abstraction α_c. In fact, we prove that a detection algorithm D that verifies the presence of a malicious behaviour in a program up to abstraction α_c, i.e., a semantic malware detector on α_c, is both sound and complete for the class of obfuscating transformations. This means that D is always able to detect programs that are infected with a conservative obfuscation of malware M (i.e., completeness), and that if D classifies a program P as infected by a malware M then a conservative obfuscation of M is actually present in program P (i.e., soundness). This means that abstraction α_c allows us to handle conservative obfuscations and their composition. Unfortunately, we are not able to provide an analogous result in the case of non-conservative transformations. Non-conservative obfuscations deeply affect program trace semantics and therefore we were not able to identify a common pattern. However, we describe some possible solutions to the design of an ad-hoc abstraction for a non-conservative obfuscation. Of course malware writers combine different obfuscating techniques to avoid detection. For this reason we show how, under certain assumptions, given the suitable abstractions for some elementary obfuscations it is possible to derive the abstraction able to deal with their composition. In this way, identifying the right abstractions for a set of elementary obfuscations we can handle also their composition. Our notion of semantic infection turns out to be quite flexible. In fact, given some specific information about the malicious behaviour that we are looking for, it is possible to weaken the original definition of semantic infection. We can say that our methodology verifies malware infection searching for a "semantic signature", while misuse detection verifies the presence of a syntactic signature. Thus, the proposed approach shares the advantages of misuse malware detection, while it is more resilient to obfuscation since it concentrates on the meaning of the malicious code and not on its syntax.

An aspect that deserves more investigation is related to the detection of non-conservative variants of a malware. Note that, by weakening the semantic notion of infection, it may be possible to find a semantic pattern that is common to a significant subset of non-conservative transformations. Our idea is to analyze the effects that typical non-conservative transformations have on program trace semantics in order to identify, if possible, some common features. If this is the case, we could further classify the family of non-conservative obfuscations and provide a suitable abstraction in order to handle such a subset. For example, the reordering of independent statements as well as the substitution of equivalent sequences of instructions could be handled by an abstraction that observes the state preceding the starting state and the state succeeding the ending state of the reordering/substitution fragment. Obviously, the point here is to define a methodology for identifying such "interesting states".

Moreover, we are interested in the investigation of the benefits that may come from the application of data mining and machine learning techniques to the systematic design of an abstraction able to handle a given obfuscating transformation. Both data mining and machine learning techniques try to discover new knowledge in large data collections, by identifying hidden patterns that a human would not be able to discover efficiently. It might be possible to specify such techniques in order to extract features that are common to different obfuscated versions of the same malware. This would provide an abstract characterization of the malicious behaviour that discards the changes made by the obfuscation while keeping the malicious intent. It is clear how such definition could be used to design an abstraction that is able to contrast a given obfuscation.

Observe that, given an obfuscation $\mathcal{O}$, the problem of systematically deriving a suitable abstraction that is able to detect all malware variants obtained through obfuscation $\mathcal{O}$, is strongly related to the problem of identifying the semantic property characterizing the obfuscating behaviour of a program transformation. Recall that a systematic methodology for deriving the most concrete property $\alpha_{\mathcal{O}}$ preserved by a program transformation $\mathcal{O}$ exists. This means that, given an obfuscation $\mathcal{O}$ and two programs P and $Q = \mathcal{O}[\![P]\!]$, then $\alpha_{\mathcal{O}}(\mathbf{S}[\![P]\!]) = \alpha_{\mathcal{O}}(\mathbf{S}[\![Q]\!])$, which guarantees completeness of the malware detector with respect to $\mathcal{O}$. The converse, which does not hold in general, would provide a more precise characterization of the obfuscating behaviour of $\mathcal{O}$ and would probably be able to guarantee the soundness of the malware detector with respect to $\mathcal{O}$. Given an abstraction $\alpha_{\mathcal{O}}$ such that $\alpha_{\mathcal{O}}(\mathbf{S}[\![P]\!]) = \alpha_{\mathcal{O}}(\mathbf{S}[\![Q]\!]) \Leftrightarrow Q = \mathcal{O}[\![P]\!]$, we have that the semantic malware detector on $\alpha_{\mathcal{O}}$ is both sound and complete with respect to $\mathcal{O}$, and abstraction $\alpha_{\mathcal{O}}$ uniquely characterizes the obfuscating power of $\mathcal{O}$. This means that abstraction $\alpha_{\mathcal{O}}$ can be used to precisely compare the power of $\mathcal{O}$ against attackers and other obfuscating techniques. Thus, the design of an abstraction $\alpha_{\mathcal{O}}$ that uniquely identifies obfuscation $\mathcal{O}$ is an impor-

tant and challenging research task both in the software protection and in the malware detection scenario. The result obtained on conservative transformations suggests to address this task by considering abstractions that characterize the semantic behaviour of classes of obfuscating transformations.

It is well known that code obfuscation is a defence technique able to defend the intellectual property of a program only for a limited period of time. In fact, given enough time, effort and determination a competent programmer is always able to defeat a given application. In some sense, a metamorphic malware solves this problem by obfuscating itself every time it infects a new machine. It may be possible to use metamorphism in order to develop a powerful defence technique. Given a program P that we want to protect, the idea is to use an obfuscating engine that replaces the current obfuscation $\mathcal{O}_1[\![P]\!]$ of the program, with a new obfuscation $\mathcal{O}_2[\![P]\!]$ with a certain frequency. In this way a malicious reverse engineer has a fixed and limited amount of time for breaking a certain obfuscation. It is clear that the set of obfuscating techniques used by the self-mutating engine have to join some "independence" property. In fact, we have to guarantee that no further information is given to an attacker who knows more than one obfuscation of P.

Software developers, as well as malware writers, typically compose different obfuscating transformations either for protecting the intellectual property of their programs or to avoid misuse detection. Thus, given two obfuscating transformations $\mathcal{O}_1$ and $\mathcal{O}_2$, it would be interesting to investigate the relationship between the obfuscating power of $\mathcal{O}_1$ and $\mathcal{O}_2$ and the one of their compositions $\mathcal{O}_1 \circ \mathcal{O}_2$ and $\mathcal{O}_2 \circ \mathcal{O}_1$. Assume that the deobfuscating engine *Deobf* knows how to recover a program when a single obfuscation is applied, namely that *Deobf* is able to handle $\mathcal{O}_1$ and $\mathcal{O}_2$. In the malware detection scenario we prove that, under certain assumptions, this means that *Deobf* can handle also their compositions. We are interested in designing a pair of obfuscating transformations for which the above result does not hold. If these transformations exist, it means that both elementary obfuscations and deobfuscations may be public, while the key for recovering the original program relies in the order in which the transformations are applied. Since there is no limit on the number of times that a transformation can be applied, this leads to a defence scheme that can be broken only "guessing" the order in which the two elementary transformations were applied. The design of such a pair of obfuscating transformations is probably related to the "independence" issue discussed above.

Another interesting field that commonly uses code obfuscation is the one of "biologically inspired diversity". In this setting, obfuscating transformations are used to generate many different versions of the same program in order to prevent malware infection [59, 128]. In fact, machines that execute the same programs are likely to be vulnerable to the same attacks. Malware exploit vulnerabilities in order to propagate and perform their damage, meaning that all

the systems sharing the same configuration will be susceptible to the same malware attacks. On the other hand, different versions of the same program are less prone to having vulnerabilities in common. This means that diverse versions of the same program will make malware infection and propagation much harder. In this setting, it would be interesting to see if our theoretical framework for code obfuscation could be used to better understand and formalized the level of security that program diversity guarantees.

References

1. L. M. Adleman. An abstract theory of computer viruses. In *Proceedings of Advances in cryptology* (*CRYPTO'88*), volume 403 of *LNCS*, 1988.
2. J. Allen, A. Christie, W. Fithen, J. McHugh, J. Packel, and E. Stoner. State of the practice in intrusion detection technologies. Technical Report 99-TR-028, ESC-99-028, Carnegie Mellon University, Software Engineering Institute, CMU/SEI, Pittsburg, PA, 2000.
3. E. G. Amoroso. *Intrusion detection: an introduction to Internet surveillance, correlation, trace back, and response*. Intrusion.net Books, 1999.
4. A. W. Appel. Deobfuscation is in NP. 2002. www.cs.princeton.edu/ appel/papers/deobfus.pdf.
5. K.R. Apt and G.D. Plotkin. Countable nondeterminism and random assignment. *J. of the ACM.*, 33(4):724–767, 1986.
6. G. Arboit. A method for watermarking java programs via opaque predicates. In *Proc. Int. Conf. Electronic Commerce Research* (*ICECR-5*), 2002.
7. D. Aucsmith. Tamper resistant software: An implementation. In *Proc. Information Hiding*, pages 317–333, 1996.
8. D. Aucsmith and G. Graunke. Tamper resistant methods and apparatus. US patent 5.892.899, Assignee: Intel Corporation, 1999.
9. A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report N01145, LAAS-CNRS, 2001.
10. S. Axelsson. Research in intrusion detection systems: A survey. Technical Report TR:98-17, Department of Computer Engineering - University of Technology - Sweden, 1998.
11. B. Barak, O. Goldreich, R. Impagliazzo, and S. Rudich. On the (im)possibility of obfuscating programs. In *Advances in Cryptology, Proc. of Crypto'01*, volume 2139 of *LNCS*, pages 1–18. Springer-Verlag, 2001.
12. J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Symposium on Requirements Engineering for Information Security*, 2001.
13. L. Briesemeister, P. A. Porras, and A. Tiwari. Model checking of worm quarantine and counter- quarantine under a group defense. Technical Report SRI-CSL-05-03, SRI International, Computer Science Laboratory, 2005.
14. D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P'06)*, 2006.
15. J. W. Bryans, M. Koutny, L. Mazarè and P. Y. A. Ryan. Opacity Generalised to Transition Systems. In *Proceedings of the 3rd International Workshop on the Formal Aspects in Security and Trust (FAST'05)*, pages 81–95,2006.

16. J. W. Bryans, M. Koutny and P. Y. A. Ryan. Modeling dynamic opacity using Petri nets with silent actions. In *Proceedings of the IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST)*, World Computer Congress, 2004, Toulouse, France. IFIP International Federation for Information Processing, Volume 173 pp. 159-172 Springer Verlag 2005
17. R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *Proc. Advances in cryptology* (*CRYPTO'97*), pages 455–469, 1997.
18. H. Chang and M. Atallah. Protecting software code by guards.
19. Y. Chen, R. Venkatesan, M. Cary, R. Pang, S sinha, and M. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive, 2002.
20. D.M. Chess and S.R. White. An undetectable computer virus. In *Virus Bulletin*, 2000.
21. F. Choen. Operating system protection through program evolution. *Computers and security*, 12(6):565–584, 1993.
22. S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proc. 4th International Information Security Conference (ISC'01)*, volume 2200 of *LNCS*, pages 144–155, 2001.
23. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium* (*Security '03*), pages 169–186, 2003.
24. M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA '04*), pages 34–44, 2004.
25. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, 2005.
26. F. Cohen. *Computer viruses*. PhD thesis, University of Southern California, 1985.
27. F. Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6(1):22–35, 1987.
28. F. Cohen. Computational aspects of computer viruses. *Computers and Security*, 8(4):325, 1989.
29. C. Collberg and K. Heffiner. The obfuscation executive. In *Proc. Information Security Conference* (*ISC'04*), volume 3225 of *LNCS*, pages 428–440, 2004.
30. C. Collberg, G. Myles, and A. H. Work. Sand mark - a tool for software protection research. *IEEE Security & Privacy*, 1(4):40–49, 2003.
31. C. Collberg and C. Thomborson. Breaking abstractions and unstructural data structures. In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '98*), pages 28–37, 1998.
32. C. Collberg and C. Thomborson. Software watermarking: models and dynamic embeddings. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* (*POPL '99*), pages 311–324. ACM Press, 1999.
33. C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Trans. Software Eng.*, pages 735–746, 2002.
34. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.
35. 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.
36. C. Consel and C. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM Symp. on Principles of Programming Languages* (*POPL '93*), pages 493–501. ACM Press, 1993.
37. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

38. P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, 1978.
39. P. Cousot. Abstract interpretation. ACM Comput. Surv., 28(2):324–328, 1996.
40. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
41. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.
42. P. Cousot and R. Cousot. Constructive versions of tarski's fixed point theorem. *Pacific J. Math.*, 82(1):43–57, 1979.
43. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.
44. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '02*), pages 178–190, New York, NY, 2002. ACM Press.
45. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97, 1978.
46. M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of the 32nd ACM Symp. on Principles of Programming Languages* (*POPL '07*), 2007.
47. M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 301–310. IEEE Computer Society Press, 2005.
48. M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proc. of the 32nd International Colloquium on Automata, Languages and Programming* (*ICALP '05*), volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336. Springer-Verlag, 2005.
49. M. Dalla Preda, M. Madou, R. Giacobazzi, and K. De Bosschere. Opaque predicate detection by abstract interpretation. In *Proc. of the 11th International Conf. on Algebraic Methodology and Software Technology* (*AMAST '06*), volume 4019 of *LNCS*, pages 81–95. Springer-Verlag, 2006.
50. B. A. Davey and H. A. Priestley. *Introduction to lattices and order.* Cambridge Press, 1990.
51. R. L. Davidson and N. Myhrvold. Method on system for generating and auditing a signature for a computer program. US patent 5.559.884, Assignee: Microsoft Corporation, 1996.
52. T. Detristan, T. Ulenspiegel, Y. Malcom, and M.S. von Underduk. Polymorphic shellcode engine using spectrum analysis, 2003.
53. S. Drape. *Obfuscation of abstract data types.* PhD thesis, The Univeristy of Oxford, 2004
54. S. Drape *An obfusction for binary trees.* TENCON 2006, to appear.
55. M. Driller. Metamorphism in practice. *29A Magazine*, 1(6), 2002.
56. T. Escamilla. Intrusion detection: Network security beyond the firewall. *John Wiley & Sons, Inc.*, 1998.
57. G. Filé and F. Ranzato. Complementation of abstract domains made easy. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming* (*JICSLP '96*), pages 348–362. The MIT Press, Cambridge, Mass., 1996.

58. S. Forrest. A sense of self for unix processes. In *Proceedings of the Symposium on Security and Privacy (S&P'96)*, pages 120–128, 1996.
59. S. Forrest, A. Somyaji and D. H. Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hoto Topics in Operating Systems*, pages 67–72, 1997.
60. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th International Static Analysis Symposium, SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 2001.
61. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.
62. G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. D. Scott. *A compendium on continuous lattices.* Springer-Verlag, 1980.
63. S. Goldwasser and Y. T. Kalai. On the impossibility of obfuscation with auxiliary input. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science* (*FOCS '05*), pages 553–562. IEEE Computer Society, 2005.
64. J. R. Gosler. Software protection: myth or reality? In *Proc. Advances in cryptology* (*CRYPTO'85*), pages 140–157, 1985.
65. Kingpin. Attacks on and Countermeasures for USB Hardware Token Devices. In *Proc. of the Fifth Nordic Workshop on Secure IT Systems Encouraging Co-operation*, pages 35–57, 2000.
66. P. Granger. Static analysis of linear congruence equality among variables of a program, 1991.
67. G. Grätzer. *General lattice theory.* Birkhäuser Verlag, Basel, Switzerland, 1978.
68. A. Gupta and R.Sekar. An approach for detecting self-propagating email using anomaly detection. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'03)*, volume 2820 of *LNCS*, pages 55–72, 2003.
69. M. H. Halstead. *Elements of software science.* Elsevier North-Holland, 1977.
70. W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. In *SIGPLAN Notices*, volume 16, pages 63–74, 1981.
71. M. Hecht. *Flow analysis of computer programs.* Elsevier, 1977.
72. A. Herzberg and S. S. Pinter. Public protection of software. *ACM transaction on computer systems*, 5(4):371–393, 1987.
73. F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1419 of *LNCS*, 1998.
74. J. Hormkovic. *Algorithmics for hard problems.* Springer-Verlag, 2002.
75. S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.
76. Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual.*
77. Munson J. C and T. M. Kohshgoftaar. Measurement of data structure complexity. *Journal of Systems Software*, 20:217–225, 1993.
78. K. A. Jackson. Intrusion detection systems (idd) product survey. Technical Report LA-UR-99-3883, Los Alamos National Laboratory, 1999.
79. N. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–504, 1996.
80. M. Jordan. Dealing with metamorphism. *Virus Bulletin*, pages 4–6, 2002.
81. L. Julus. Metamorphism. *29A Magazine*, 1(5), 2000.
82. G. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st ACM Symp. on Principles of Programming Languages* (*POPL '73*). ACM Press, 1973.
83. H.-A. Kim and B. Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

84. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *LNCS*, pages 174–187, 2005.
85. C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs using execution monitoring. In *Proceedings of the 10th Computer Security Application Conference*, 1994.
86. C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 175–187, 1997.
87. J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the 10th ACM SIGKDD International conference on Knowledge Discovery and Data Mining (KDD'04)*, pages 470–478, 2004.
88. S. Kumar. *Classification and detection of computer intrusions*. PhD thesis, Department of Computer Science, Purdue University, 1995.
89. S. Kumar and E. H. Spaffored. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1995.
90. Y. Lakhnech and L. Mazar. Probabilistic opacity for a passive adversary and its application to Chaum's voting scheme. Technical report TR-2005-4, Verimag, 2005
91. A. Lakhotia and P. K. Singh. Challenges in getting "formal" with viruses. In *Virus Bulletin*, 2000.
92. W. Landi and B. G. Ryder. A safe approximate algorithm for inter-procedural pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '92*), pages 235–248, 1992.
93. W. Lee, R. A. Nimbalkar, K. K. Yee, S. B. Patil, P. H. Desai, T. T. Tran, and S. J. Stolfo. A data mining and cidf based approach for detecting novel and distributed intrusions. volume 1907 of *LNCS*, pages 49–65, 2000.
94. W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, pages 79–93, 1998.
95. W. Lee, S. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P'99)*, pages 120–132, 1999.
96. W. J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Systems, Man and Cybernetics (SMC) Workshop on Information Assurance (IAW'05)*, pages 64–71, 2005.
97. Z. Li and A. Das. Visualizing and identifying intrusion context from system call trace. In *Proceedings of the 20th Annual Computer Security Applications Conference*, 2004.
98. Z. Li, A. Das, and J. Zhou. Theoretical basis for intrusion detection. In *Proceedings of the 6th IEEE Information Assurance Workshop (IAW)*, 2005.
99. Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on computer and Communications Security (CCS'05)*, pages 213–222, 2005.
100. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Computer Security Symposium* (*CSS '03*), pages 290–299, 2003.
101. R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: A malicious code filter. *Computers & Security*, 14:541–566, 1995.
102. B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *Proceedings of Eurocrypt 2004*, 2004. citeseer.ist.psu.edu/lynn04positive.html.
103. M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *Proc. Internat. Workshop on Information Security Applications* (*WISA'05*), volume 3786 of *LNCS*, pages 194–206, 2005.

104. M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proc. 5th ACM Workshop on Digital Rights Management (DRM'05)*, 2005.
105. M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de)obfuscation tool. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, pages 140–144, 2006.
106. J. Maebe, M. Ronsse, and K. De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Proc. 4th Workshop on Binary Translation (WBT'02)*, 2002.
107. A. Majumdar and C. Thomborson. Securing mobile agents control flow using opaque predicates. In *Proc. 9th Int. Conf. Knowledge-Based Intelligent Information and Engineering Systems (KES'05)*, 2005.
108. A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proc. 29th Australasian Computer Science Conference (ACSC'06)*, volume 48 of *CRPIT*, pages 187–196, 2006.
109. J. Marciniak editor. *Encyclopedia of software engineering.* J. Wiley & Sons, Inc, 1994.
110. G. McGraw and G. Morrisett. Attacking malicious code: Report to the Infosec research council. *IEEE Software*, 17(5):33–41, 2000.
111. J. McHugh. Intrusion and intrusion detection. *International Journal of Information Security*, 1(1):14–35, 2001.
112. C. Michael, G. McGraw, M. Schatz, and C. Walton. Genetic algorithms for dynamic test data generation. In *Proc. ASE'97*, pages 307–308, 1997.
113. A. Minè. The octagon abstract domain. In *Proc. Analysis, Slicing and Transformation (AST'01)*, pages 310–319, 2001.
114. P. Morley. Processing virus collections. In *Proceedings of Virus Bulletin*, pages 129–134, Prague, Czech Republic, 2001. Virus Bulletin.
115. S. A. Moskowitz and M. Cooperman. Method for stega-cipher protection of computer code. US patent 5.745.569, Assignee: The Dice Company, 1996.
116. G. Myles and C. Collberg. Software watermarking via opaque predicates: implementation, analysis, and attacks. In *Proc. Int. Conf. Electronic Commerce Research (ICECR-7)*, 2004.
117. F. Nielson, H. Nielson and C. Hankin *Principles of Program Analysis.* Springer Verlag, 1999.
118. C. Nachenberg. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers*, XXX:1–13, 1996.
119. C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, 1997.
120. J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P'05)*, pages 226–241, 2005.
121. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS & P'05)*, 2005.
122. S. Northcutt, M. Cooper, M. Fearnow, and K. Frederick. *Intrusion signature and analysis.* New Riders, SANS GIAC, 2001.
123. T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Sftware obfuscation on a theoretical basis and its implementation. *IEEE Trans. Fundamentals*, E86-A(1), 2003.
124. E. I. Oviedo. Control flow, data flow and programmers complexity. In *Proc. of COMPSAC 80*, pages 146–152. Chicago, IL, 1980.
125. R. Paige. Future directions in program transformations. *ACM SIGPLAN Not.*, 32(1):94–97, 1997.

126. J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th IEEE Annual Security Applications Conference* (*ACSAC '00*), pages 308–316, 2000.
127. A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *STTT*, 2(2):192–201, 1998.
128. R. Pucella and F. B. Schneider. Independence from Obfuscation: A Semantic Framework for Diversity. In *Proceedings of the 19th IEEE Computer Security Foundation Workshop*, pages 230–241,2006.
129. Rajaat. Polymorphism. *29A Magazine*, 1(3), 1999.
130. G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1997.
131. M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the 11th Systems Administration Conference (LISA), USENIX*, pages 1–8, 1997.
132. M. Samamura. *Expanded threat list and virus encyclopedia*. Symantec Antivirus Research Center, chapter W95.CIH, 1998.
133. P. Samuelson. Reverse-engineering someone else's software: Is it legal? *IEEE Software*, pages 90–96, 1990.
134. B. Schwarz, S. Debray, and G. Andrews. PLTO: A link-time optimizer for the intel ia-32 architecture. In *Proc. Workshop on Binary Translation (WBT'01)*, 2001.
135. P. Singh and A. Lakhotia. Static verification of worm and virus behaviour in binary executables using model checking. In *Proceedings of the 4th IEEE Information Assurance Workshop*, 2003.
136. S. R. Snapp, S. E. Smaha, D. M. Teak, and T. Grance. The DIDS (distributed intrusion detection system) prototype. In *USENIX Conference*, pages 227–233, 1992.
137. D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):159–176, 2003.
138. P. A. Suhler, N. Bagherzadeh, M. Marlek, and N. Iscoe. Software authorization systems. *IEEE Software*, 3(5):34–41, 1986.
139. Symantec Corporation. Symantec Internet security threat report: Trends for january 06–june 06. X, 2006.
140. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.
141. P. Szor and P. Ferrie. Hunting for metamorphic. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*, pages 123 – 144, 2001.
142. A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–310, 1955.
143. A. M. Turing. On computable numbers, with an application to the entscheidungs problem. In *Proceedings London Math. Soc.*, volume 2, pages 230–265, 1936.
144. S. K. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th. IEEE Working Conference on Reverse Egineering* (*WCRE '05*), 2005.
145. H. Vaccaro and G. Liepins. Detection of anomalous computer sessions activity. In *Proceedings of the Symposium on Security and Privacy (S&P'89)*, pages 280–289, 1989.
146. H. P. Van Vliet. Crema – the java obfuscator. 1996.
147. C. Wang. *A security architecture for survivability mechanisms*. PhD thesis, University of Virginia, 2000.
148. C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: obstructing static analysis of programs. Technical report CS-2000-12, Department of Computer Science, University of Virginia, 2000.
149. M. Ward. The closure operators of a lattice. *Annals of Mathematics*, 43(2):191–196, 1942.
150. M. Webster. Algebraic specification of computer viruses and their environments. In Peter Mosses, John Power, and Monika Seisenberger, editors, *Selected Papers from the First*

Conference on Algebra and Coalgebra in Computer Science Young Researchers Workshop (CALCO-jnr 2005). University of Wales Swansea Computer Science Report Series CSR 18-2005, pages 99–113, 2005.

151. H. Wee. On obfuscating point functions. In *Proc. ACM STOC 2005*, pages 523–532, 2005.
152. M. Weiser. Program slicing. *IEEE Trans. Software Engineering SE*, 10(4):352–357, 1984.
153. J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th Conference on Computer and Communication Security (CCS'05)*, pages 223–234, 2005.
154. H. Yang and Y. Sun. Reverse engineering and reusing COBOL programs: A program transformation approach. In *IWFM '97 Electronic Workshop in Computing*, 1997.
155. z0mbie. Automated reverse engineering: Mistfall engine. Published online at `http://www.madchat.org//vxdevl/papers/vxers/Z0mbie/autorev.txt`,.
156. z0mbie. Real permutating engine. Published online at `http://vx.netlux.org/vx.php?id=er05` (last accessed on Sep. 29, 2006).
157. W. Zhu, C. Thomborson, and F. Wang. Obfuscate arrays by homomorphic functions. In *Special Session on Computer Security and Data Privacy in IEEE GrC 2006*, pages 770–773, 2006.

Sommario

Un offuscatore trasforma programmi in modo da preservarne la funzionalità e garantendo allo stesso tempo che i programmi trasformati risultino più complessi, ovvero più difficili da capire, rispetto a quelli originali. La protezione della proprietà intellettuale del codice e l'identificazione di programmi maleintezionati, chiamati nel seguito malware, rappresentano due dei maggiori campi di applicazione dell'offuscamento di codice. L'offuscamento è infatti comunemente usato dagli scrittori di programmi per difendere la proprietà intellettuale del proprio lavoro da possibili attacchi. Rendere i programmi più difficili da capire permette infatti di contrastare il malicious reverse engineering, ovvero l'analisi di programmi a fini illeciti. D'altra parte, gli scrittori di malware, solitamente chiamati hackers, fanno uso delle tecniche di offuscamento per impedire ai rilevatori di malware di identificarli. Gran parte degli algoritmi per il rilevamento di malware si basano su aspetti puramente sintattici dei programmi, ovvero sul modo in cui i programmi maliziosi sono scritti e non sul loro comportamento. Chiaramente questa caratteristica dei rilevatori fa si che l'identificazione di un malware sia fortemente sensibile anche a minime variazioni della loro sintassi. Nell'ambito della protezione del software si è interessati allo sviluppo di tecniche di offuscamento sempre più sofisticate al fine di proteggere i programmi dal maggior numero possibile di attacchi alla loro proprietà intellettuale. D'altra parte, per quanto concerne il rilevamento di malware, è importante sviluppare algoritmi avanzati di identificazione di codice maleintenzionato, al fine di individuarne la più vasta varietà di versioni, ovvero di offuscamenti. Chiaramente, entrambe le tipologie di attacco descritte rappresentano un importante pericolo per la sicurezza delle reti di computers.

In questo lavoro ci siamo interessati ad ebtrambi i problemi di sicurezza sopra descritti. In particolare, proponiamo un approccio formale all'offuscamento di codice basato sulla semantica dei programmi e sulla teoria dell'interpretazione astratta. La struttura teorica che introduciamo risulta utile al fine di arginare alcuni noti svantaggi della protezione del codice attraverso l'offuscamento, e per

migliorare le esistenti tecniche di rilevamento dei malware. Uno dei maggiori svantaggi dell'offuscamento di codice come tecnica di protezione della prorietà intellettuale dei programmi, è dato dalla mancanza di solide base teoriche, che rendende difficile la cerificazione dell'efficienza di questi approcci nel difendere la proprietà dei programmi. Inorte, per poter ideare algoritmi di rilevamento di programmi maleintenzionati che siano in grado di gestire l'offuscamento è necessario concentrarsi sulla semantica dei programmi e non solo sul loro aspetto sintattico.

Uno dei punti cruciali del nostro approccio formale all'offuscamento di codice è dato dall'introduzione di una nozione di offuscamento basata sulla semantica di programmi. In particolare, seguendo la nostra definizione, ogni trasformazione T di programmi può essre vista come un potenziale offuscamento, dove il grado di complessità aggiunto al programma dalla trasformazione è espresso in termini della più concreta proprità semantica che T preserva. Ovvero, tale prorietà esprime ciò che la trasformazione T preserva del comportamento del programma originale, e quindi caratterizza anche ciò che viene nascosto e che non è possibile osservare dopo l'offuscamento. Le tecniche di offuscamento, cosí come il reverse engineering, cominciano solitamente con un' analisi statica del programma e possono quindi essere specificate come astrazioni della semantica dei programmi. Questa osservzione ci porta a modellare gli attaccanti, nell'ambito della protezione del software, come astrazioni della semantica di programmi. In particolare, un attaccante viene modellato dall'astrazione che esprime in modo preciso l'informazione, ovvero le proprietà semantiche, che l'attaccante è in grado di osservare di un programma. È quindi possibile confrontare il grado di astrazione di un attaccante A con quello della più concreta priprietà preservata da un offuscamento T e capire se la tecnica T è in grado di proteggere i programmi dall'attacco A. Seguendo lo stesso principio, diverse techniche di offuscamento possono essere confrontate rispetto al grado di sicurezza che garantiscono, e diversi attacchi rispetto alla loro efficacia nello sconfiggere una data protezione. Per validare la nostra struttura formale, l'abbiamo applicata ad una nota tecnica di offuscamento di controllo che trasforma il flusso di controllo del programma originale inserendo dei predicati opachi.

Abbiamo osservato come gli offuscamenti siano delle trasformazioni di programmi che preservano un'astrazione della semantica. Infatti, diverse versioni offuscate di un malware sono accomunate (almeno) dallo stesso intento malizioso, ovvero presentano lo stesso comportamento mleintenzionato, pur esprimedolo attraverso diverse forme sintattiche. Il nostro approccio formale al rilevamento di programmi maleintenzionati, prevede l'utilizzo della semantica per modellare sia i programmi che i malware e l'impiego di astrazioni semantiche per nascondere i dettagli che vengono modificati in fase di offuscamento. Quindi, dato un offuscamento T, si vuole individuare l'astrazione semantica, ovvero le proprietà semantiche, che accomunano la semantica del malware M con la semantica del

suo offuscamento $T(M)$. È chairo che, dato un offuscamento T, l'identificazione di una proprietà semantica con le caratteristiche sopra descitte rappresenta uno dei punti più delicati dell'approccio proposto. A questo proposito, siamo in grado di fornire un'astazione semantica in grado di contrastare un'interessante classe di offuscamenti comunemente usati, e diverse strategie per dedurre l'astazione adeguata per le trasformazioni che non appartengono a questa classe.

Si osservi che, dato un rilevatore di programmi maleintenzionati D, analizzando come opera sulla semantica di programmi, è sempre possibile definirne la controparte semantica. Traducendo, come proposto, sia i rilevatori di malware che gli offuscamenti in operatori semantici, è possibile dimostrare quali offuscamenti un rilevatore è in grado di gestire e confrontare l'efficienza delle diverse tecniche di offuscamento. Quindi, la nostra struttutra semantica fornisce un ambiente formale dove coloro che sviluppano algoritmi di rilevamento di programmi maleintenzionati possono dimostrare l'efficienza dei loro prodotti.