Code and Proof Obfuscation

François Dupressoir
under the supervision of David Pichardie

February-June 2008

This document reports the work done in the LANDE team, at the IRISA, in Rennes for partial fulfillment of a Master's degree in Computer Science at the École Normale Supérieure de Cachan – Antenne de Bretagne.

Code and Proof Obfuscation – Abstract

François Dupressoir

Ensuring safety properties when running a program is often a priority on embedded systems. However, intellectual property protection often requires some secrecy about the inner workings of a program. We propose a new obfuscation technique, based on an abstract interpretation view of opaque predicates, that interfaces well with a given proof-carrying code technique. We show that the presence of the proof certificate does not make deobfuscation easier, and that the obfuscation can, if done properly, keep the proof certificate sound and complete with regard to the proved property.

Contents

Introduction **3**

Ubiquity and Security . . . 3

Ubiquity and Intellectual Property . . . 3

Ubiquity and Host Security . . . 4

Providing Protection on Both Sides . . . 5

Using Abstract Interpretation . . . 6

The Thesis . . . 6

1 Preliminaries **7**

1.1 Static Analysis . . . 7

1.1.1 Principles . . . 7

1.1.2 A Formal Point of View . . . 8

1.2 Abstract Interpretation . . . 10

1.2.1 Abstractions, Closure Operators, and Galois Connections . . . 10

1.2.2 Approximate Analysis . . . 11

1.2.3 Lattice of Abstractions . . . 13

2 State of The Art **15**

2.1 Proof-Carrying Code . . . 15

2.1.1 Generalities . . . 15

2.1.2 LANDE's PCC architecture . . . 16

2.2 Obfuscation . . . 17

2.2.1 Syntactic Obfuscation . . . 18

2.2.2 Semantic Obfuscation . . . 19

Formalizing the Problem **21**

3 A Semantic Solution **23**

3.1 Semantic Control-Flow Obfuscation 23
3.1.1 Semantic Detection of Opaque Predicates 24
3.1.2 Soundness of the Abstract Test 25
3.1.3 Completeness of the Abstract Test 26
3.2 Preliminary Study: Opaque Predicates 26
3.2.1 Opaque Predicates and Invariants 26
3.2.2 Opaque Predicates and Invariant Composition 27
3.3 A Solution 28
3.3.1 Building Opaque Predicates 29

4 An Abstract Point of View **33**
4.1 Partitionning the Lattice of Abstractions 33
4.1.1 Monotonicity of Computational Power 34
4.1.2 Direct Results 34
4.1.3 Open Leads 35
4.2 Making our Opaque Predicates Stealthy 36
4.2.1 Masking Standard Predicates 37
4.2.2 Unmasking Opaque Predicates 37

Conclusion **39**

A A `While` language **41**
A.1 Syntax 41
A.1.1 Common Shorthands 41
A.2 Semantics 42
A.2.1 Control-Flow Graph 42
A.2.2 Collecting Semantic 44
A.3 Non-Deterministic Extension: ND-`While` 45

Bibliography **49**

Introduction

Ubiquity and Security

With the increasing number of different computer systems now on the market, portability has become an asset in the software industry. For many applications (cell phone and PDA-aimed software), it is simply not possible for program ditributors to issue one version per platform. This is why bytecode-distributed languages, such as Java, or the .NET-based languages, are so widespread nowadays. However, bytecode distribution also has major drawbacks, both for the program intellectual property owner, and for its user.

Ubiquity and Intellectual Property

Ubiquity and the important increase in the differences between computing platforms and requirements lead to the development of bytecode-distributed programs. Bytecode, as an intermediate representation of programs, retains most of the semantic information of the program, allowing for easier reverse-engineering operations. This can be unacceptable to a software provider for many reasons, one of which being the respect of his intellectual property. Different solutions have been suggested to protect Bytecode software from reverse-engineering:

Software cryptography The program is distributed as an encrypted package and is decrypted before being run, or is run in a secure virtual machine. This prevents direct observation of the source code, but does not, of course, prevent a well-equipped attacker from observing the sequence of instructions at processor-level, thus inferring the program's bytecode representation. Moreover, current cryptographic techniques have important overheads that would hinder program performance.

Hardware cryptography The program is distributed as an encrypted package and is run on a specially-designed piece of hardware (either a dedicated host, or a dongle). Another alternative could be to encrypt only sensitive parts of the program to be run securely. The overhead of such a method is important, not necessarily in time (even though transmitting to a dongle can be quite costly), but also in hardware requirements, thus hindering portability.

Obfuscation Obfuscation has always been more or less used and never formalized. It consists in scrambling the data-flow and control-flow graphs by modifying the low-level syntax of the program without modifying its denotational semantics (input/ouptut relations). Various complementary techniques exist and are used, especially on Java bytecode ([CTL97]). Obfuscation does not*prevent* reverse engineering, but does make it more costly in time and ressources by making static analysis less precise. However, the low overhead it incurs in the program, and the absence of hardware or trust requirements, make it the most widely used technique in intellectual property protection. We will say more about obfuscation in section 2.2.

Ubiquity and Host Security

On the other hand, embedded systems are often very sensitive to new software components and require some safety properties on those to ensure their own stability: insertion of malicious –or even simply faulty– code in a piece of software destined for use in one of those systems might cause the entire system to crash. Thus, we need to provide the host running the sensitive code with a way of proving statically that the latter is safe to run. Several possibilities exist for this purpose:

Cryptographic signature Cryptographic signature can be used to ensure that a given program does come from a certain trusted source. However, such a technique requires a trust relation between the program maker and the program user, as well as a trusted third-party to enable the cryptographic signature protocol.

Dynamic checks It is possible –once a set of *dangerous instructions* is defined– to check dynamically that these operations are executed safely, by observing the memory state every time they happen. The overhead of such checks (such as the array bounds checks implemented in the JVM) is quite important, however, making them rather costly on mobile systems that typically have little ressources.

Proof of program properties It is possible to semi-automatically prove many basic properties on structured programs. However, the semi-automatic aspect of this proof, and the high ressource requirements of the usual program proof techniques make such an approach impossible to perform on the client's side. Moreover, the client will not usually have acces to the structured form of the program, on which the proof can be done.

Proof-Carrying Code (PCC) The solution was proposed by Necula [Nec97]: the proof can be computed by the code provider, then sent in a compressed form along with the program, and checked by the running host before the program is executed. This approach does not require any trust relation, and we will see in section 2.1 different ways of reducing the overhead due to the transmission of the proof certificate, and to reduce the ressource requirements of the proof-checking performed by the client.

Providing Protection on Both Sides

Those two goals are dual, and achieving them together without them hindering one another does not seem easy. However, protecting the recipient of a program while keeping in mind the protection of intellectual property has been studied from a cryptographic point of view ([CSV07]), as pointed out by an anonymous referee. However, this approach is valid only in an industrial setting, where extra infrastructure or overhead is affordable. Indeed, the authors propose an approach where the code issuer is provided with a "hostage proof server", called *amanat* by the client, who also provide a formal specification of the wanted program. All communications between the client and the *amanat* are filtered by the issuer, who does not have physical or remote access to it. In that situation, which the authors argue is easy to produce with various levels of confidence and cost, the code issuer can trust the *amanat* not to divulge the source to the outside world, and the code recipient can trust the *amanat* to provide him with a valid conformance proof.
However, it is clear that:

1. This approach is only applicable to mobile computing with an important ressource overhead since it relies on a third-party requiring a very high level of trust.
2. This approach is not applicable to bytecode-distributed languages, for which not providing the source is not sufficient to ensure the protection of intellectual property.

Other than the previous approach, we do not know of any other attempt at providing a proof that a program verifies certain properties while protecting the intellectual property of the program rights' owner. Chapters 3 and 4 will present the technique we suggest, the first in a rather hands-on way, and the latter from a more abstract point of view, aiming at precising and formalizing some of the ideas express in the previous chapter.

Using Abstract Interpretation

Since we want to provide formal proofs that our PCC technique is not in any way made unsound by the application of our obfuscation technique, we will need a common formal framework to study both operations. As PCC and obfuscation both require some amount of preliminary static analysis (see chapter 2), abstract interpretation, which has been used to formalize diverse static analyses seems to be a good starting point. Moreover, it is not the first time absract interpretation has been used to formalize –or define a formal framework for– intellectual property-related issues, as Cousot himself used abstract interpretation for software watermarking ([CC04a]), and Mila Dalla Preda ([Dal07]) studied obfuscation using abstract interpretation (more will be said on this in section 2.2).

The Thesis

At the light of this short introduction to the different concepts involved, we can now formulate the thesis that drove my work in the Lande team:

> It is possible to enable a proof-carrying code technique on Java bytecode while protecting the program from reverse-engineering and other attempts at breaking the intellectual property, and we will attempt to do so using the formal framework provided by abstract interpretation. We will be paying particular attention not to make deobfuscation any easier for an attacker with the transmitted proof certificates.

Chapter 1

Preliminaries

This section introduces the basic concepts of static analysis and abstract interpretation.

1.1 Static Analysis

Ever since compilers existed, and programs got too complicated to be studied directly by a human being, static analysis has been used automatically to find out some properties of a program without observing its run-time behaviour. The use of static analysis has been generalized from code optimization to error detection (null pointer analysis, array bounds...) and invariant extraction for program proof.

1.1.1 Principles

The basic principle of static analysis is to execute the program on *semantic properties*, instead of values. A *property* is a subset of the set of states. For example, in a program that has two integer variables, the set of states will be $State = \mathbb{Z}^2$, and properties will be elements of $\mathcal{P}(State)$, the powerset of $\mathbb{Z}^2$. Figure 1.1 shows a quick example of a classic analysis on a short program that we will study repeatedly throughout this document.

1: `x:=0;`		
$\{(0,\bot)\}$	$\{(0,\bot)\}$	$\{(0,\bot)\}$
2: `y:=0;`		
$\{(0,0)\}$	$\{(0,0),(1,0),(1,2)\}$	$\{(0,0),(1,0),(1,2),\ldots\}$
3: `while (x<6) {`		
$\{(0,0)\}$	$\{(0,0),(1,0),(1,2)\}$	$\{(0,0),(1,0),(1,2),\ldots\}$
4: `if (?<?) {`		
$\{(0,0)\}$	$\{(0,0),(1,0),(1,2)\}$	$\{(0,0),(1,0),(1,2),\ldots\}$
5: `y:=y+2;`		
$\{(0,2)\}$	$\{(0,2),(1,2),(1,4)\}$	$\{(0,2),(1,2),(1,4),\ldots\}$
`};`		
$\{(0,0),(0,2)\}$	$\{(0,0),(0,2),(1,0),(1,2),(1,4)\}$	$\{(0,0),(0,2),(1,0),(1,2),(1,4),\ldots\}$
6: `x:=x+1;`		
$\{(1,0),(1,2)\}$	$\{(1,0),(1,2),(2,0),(2,2),(2,4)\}$	$\{(1,0),(1,2),(2,0),(2,2),(2,4),\ldots\}$
`}`		

Figure 1.1: An example of a classic static analysis.

1.1.2 A Formal Point of View

More formally, static analyses compute the collecting semantic of a given program by resolving a fixed-point equation on the considered program points.[1] The example of figure 1.1 can be formalized as the equation system shown in figure 1.2.

Elements of Lattice Theory

To solve such equation systems, one often considers the property domain as a *complete lattice*. The semantic operators being monotone and continuous, one can then use *Kleene's theorem* to constructively characterize the least fixed-point of function F –that is, the least solution to the system.

Definition 1 *Complete lattice*
A partially ordered set $(L, \leq)$ *is a* complete lattice *if every subset* S *of* L *has both a* greatest lower bound $\bigwedge A$ *and a* least upper bound $\bigvee A$ *in* L*.*

[1]See appendix A.2.2 for the definition of the collecting semantics of our language

```
      X1 = {(⊥,⊥)}
1:  x:=0;
      X2 = [[x := 0]]X1
2:  y:=0;
      X3 = [[y := 0]]X2 ∪ X6
3:  while (x<6) {
      X4 = [[x < 6]]↓X3
4:    if (?<?)  {
      X5 = [[? <?]]↓X4
5:        y:=y+2;
      X5' = [[y := y + 2]]X5
      };
      X6 = X5 ∪ X5'
6:    x:=x+1;
      X7 = [[x := x + 1]]X6
    }
      Xend = X3 ∪ X7
```

Figure 1.2: The fixed-point equation corresponding to example 1.1.

This definition is given here only to give, without further details, an intuition on why this restricted framework is often sufficient to model semantic properties: as we have seen properties are in fact subsets of the program's real domain. Any powerset, equipped with the inclusion order, is a complete lattice with *glb* $\bigcap$ (intersection) and *lub* $\bigcup$ (inclusion), least element $\emptyset$ and greatest element the entire set.

Theorem 1 *Kleene's Theorem*
Let $(L, \bigcap, \bigcup, \bot, \top)$ *be a complete lattice, and* $f : L \rightarrow L$ *be continuous and monotone. Then* f *has a least-fixed point* $lfp(f)$ *and* $lfp(f) = \bigcup_n f^n(\bot)$.

Clearly, in this context, all fixed-point computations on domains that do not have any infinite ascending chains will eventually terminate by iterating F. It is not the case in general, however, that all conditions are verified: complete lattices that verify the ascending chain condition are rare, even in the abstract world. Moreover, even on those that do verify the ascending chain condition (and some may argue that all lattices considered in "real" computer science are since they are finite), convergence time can be too important to practically use Kleene's characterization of the least-fixed point. Moreover, the X_k

themselves are often infinite, even on simple examples such as the one given in figure 1.2.

1.2 Abstract Interpretation

The solution is to perform the computation on over-approximations of properties and operations that have a finite description, yielding an over-approximation of the final result. This idea was formalized by the Cousots in 1977 ([CC77]) as abstract interpretation.

1.2.1 Abstractions, Closure Operators, and Galois Connections

The key idea of abstract interpretation is to group properties that share a common characteristic –relevant to the analysis we want to perform– together, abstracting away from irrelevant information. For example, if all we want is to prove that no computation can overflow the integer capacity, we can use the *Int* abstraction of intervals (see figure 1.3). Let us formalize this notion of abstraction.

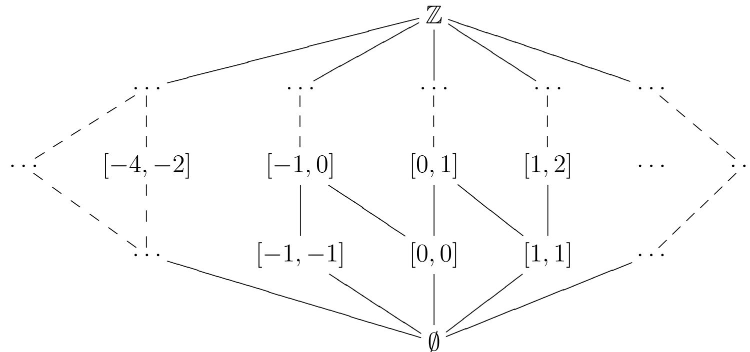


Figure 1.3: The *Int* abstraction

Abstractions

Since we want our analysis to be sound (*i.e.* to find all errors), we need to abstract in a conservative manner, all properties being represented by an over-approximation of it. Moreover, we would like to have a unique best abstract property for each concrete property. Given a concrete complete lattice $(\mathcal{D}, \subseteq, \bigcap, \bigcap)$, an abstraction of $\mathcal{D}$ is thus a sublattice $\overline{\mathcal{D}}$ with induced order and bounds such that $\forall p \in \mathcal{D}, \left\{\overline{p} \in \overline{\mathcal{D}} | p \subseteq \overline{p}\right\}$ has a

least element. This is called a Moore family of $\mathcal{D}$.

A better characterization of such abstractions, however, equivalent to the Moore family formulation, using (upper) closure operators.

Definition 2 *Closure Operator*
Let $(\mathcal{D}, \sqsubseteq, \bigsqcap, \bigsqcup)$ *be a complete lattice. A function* $\rho : \mathcal{D} \rightarrow \mathcal{D}$ *is an upper closure operator iff* ρ *is:*

- *monotone* $(\forall x, y \in \mathcal{D}, x \sqsubseteq y \Rightarrow \rho(x) \sqsubseteq \rho(y))$*,*
- *extensive* $(\forall x \in \mathcal{D}, x \sqsubseteq \rho(x))$*,*
- *idempotent* $(\rho \circ \rho \dot{=} \rho)$*.*

Theorem 2 *Upper Closure Operator Characterization of Abstractions*
A sublattice $\overline{\mathcal{D}}$ *of* $\mathcal{D}$ *is an abstraction of* $\mathcal{D}$ *iff there exists an upper closure operator* ρ *such that* $\overline{\mathcal{D}} = \{\rho(p) | p \in \mathcal{D}\}$*.*

We will often denote by ρ the abstraction as well as the closure operator and its image. Abstractions are often characterized further using Galois connections and Galois insertions but we will make no direct use of this characterization in this report. However, if anyone wanted to read more about abstract interpretation, they should start by reading the one of the very good introductions to abstract interpretation written by Cousot [CC92, CC04b].

1.2.2 Approximate Analysis

Once a good abstraction has been chosen (we will here follow on the examples using the *Sign* abstraction (figure 1.4) composed with a cartesian abstraction that is not described here), we can define an abstract operator f^i_ρ for every concrete operator f^i in a sound fashion, by ensuring that $\rho \circ f^i \dot{\subseteq} f^i_\rho \circ \rho$. The system of concrete semantic equations can then be abstracted, like it is done in figure 1.5 with the example program. We will not give here the definitions of the abstract operators, but the reader can easily imagine what their definition looks like. Figure 1.6 shows the abstract solution to the system given in the previous figure. Even with such a simple abstraction, we already managed to deduce that both variables x and y stay positive throughout the execution of our program, whereas we could not deduce anything using the concrete static analysis.
On "big" lattices and lattices that do not verify the ascending condition chain, one can use widening and narrowing heuristics to accelerate the convergence, or to ensure

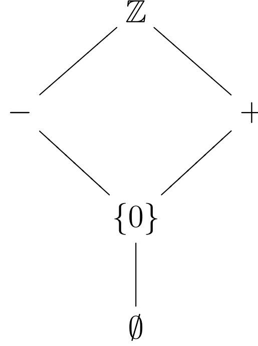


Figure 1.4: The *Sign* abstraction

$X_1^\rho = \{(\bot, \bot)^\rho\}$

1: `x:=0;`

$X_2^\rho = [\![x := 0]\!]_\rho X_1^\rho$

2: `y:=0;`

$X_3^\rho = [\![y := 0]\!]_\rho X_2^\rho \cup^\rho X_6^\rho$

3: `while (x<6) {`

$X_4^\rho = [\![x < 6]\!]_\rho^\downarrow X_3^\rho$

4: `if (?<?) {`

$X_5^\rho = [\![? <?]\!]_\rho^\downarrow X_4^\rho$

5: `y:=y+2;`

$X_5'^\rho = [\![y := y + 2]\!]_\rho X_5^\rho$

`};`

$X_6^\rho = X_5^\rho \cup^\rho X_5'^\rho$

6: `x:=x+1;`

$X_7^\rho = [\![x := x + 1]\!]_\rho X_6^\rho$

`}`

$X_{end}^\rho = X_3^\rho \cup^\rho X_7^\rho$

Figure 1.5: The abstract fixed-point equations

termination, at the expense of precision. The analysis then yields an over-approximation of the abstract fixpoint.

$X_1^\rho = (\emptyset, \emptyset)$

1: `x:=0;`

$X_2^\rho = (\{0\}, \emptyset)$

2: `y:=0;`

$X_3^\rho = (+, +)$

3: `while (x<6) {`

$X_4^\rho = (+, +)$

4: `if (?<?) {`

$X_5^\rho = (+, +)$

5: `y:=y+2;`

$X_5'^\rho = (+, +)$

`};`

$X_6^\rho = (+, +)$

6: `x:=x+1;`

$X_7^\rho = (+, +)$

`}`

$X_{end}^\rho = (+, +)$

Figure 1.6: The abstract fixed-point

1.2.3 Lattice of Abstractions

An important point in the formalism is that abstractions can be compared with respect to their relative precision. For example, the *Sign* abstraction is less precise (or more abstract) than the *Int* abstraction, which is itself more abstract then the *Poly* abstraction. Cousot used this idea to show that the different classes of semantics that are commonly used can be ordered into a hierarchy [Cou97]. The lower in the lattice of abstractions, the more properties the abstraction computes, but also the harder to compute it is.

Reduced Product

Since they are ordered into a complete lattice, you can take the least upper bound of two abstractions (this is simply their union), but also their greatest lower bound. This operation is called reduced product. It allows one to combine two abstractions to define a third one that is at least as good as their simple composition in terms of what properties it can compute. However, there is in general no definition of the best abstract operators

on the reduced product of two abstractions in terms of the best abstract operators on the two original abstractions.

Chapter 2

State of The Art

In this section, we will recall briefly the state of the art on proof-carrying code and obfuscation, already discussed in the preliminary bibliography report.[1] We will however insist here on the semantic approaches to both these problems, and will develop the Lande solution for proof-carrying code in more details, since it will be the basic starting point for our suggested solution.

2.1 Proof-Carrying Code

As stated in the introduction, Necula introduced PCC [Nec97] as a means of allowing a trusted platform to run untrusted software while ensuring some safety properties on its behaviour, and still retaining interesting performance compared to other approaches.

2.1.1 Generalities

The basic principle of PCC is that program issuer provides, along with the program $\mathbb{P}$, an untrusted certificate $\mathcal{C}$. Before running $\mathbb{P}$, the client will check that:

1. $\mathcal{C}$ indeed corresponds to $\mathbb{P}$;
2. and $\mathcal{C}$ proves that $\mathbb{P}$ verifies the client's safety policy.

[1] Available at ftp://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2008/biblio_Dupressoir_Francois.pdf

The most important thing is that the checking process has to be automatic, and light in ressources and time, since its aim is to be run on the client's side, which is typically an embedded system. The certificate generation process, however, can be as complicated as wanted, and is not even necessarily automatic. Necula identified two main problems to solve to give an efficient PCC solution:

Certificate size The certificate needs to be small enough so that the transmission overhead is low, since communications are often expensive on the considered systems (*e.g.* mobile phones).

Proof-checking simplicity The proof-checking process needs to be straightforward and to use as little ressources as possible. It is also wanted that the checker is the same for all certificates and does not depend on the program, but only on the safety policy.

Several solutions were proposed in this framework, none of them formal enough to formally prove the safety of the proof process.

2.1.2 Lande's PCC architecture

This is what motivated the work in the Lande team to use previous work by Albert *et al.* [APH04], that presented an abstract interpretation-based PCC architecture for logic programs, and develop a PCC architecture for Java bytecode that could be formally certified in Coq [BJPT07].

Basic Idea

The basic idea behind using abstract interpretation to perform PCC is that we can compute a sound approximation $S^\#$ of the program's semantic that is focused –using the abstraction– on proving a certain safety property ϕ. This sound approximation can then be transmitted, after some sort of compression, along with the program, and all the checker needs to do is to compute the program's abstract fixed-point equation system $F^\#$ and verify, in one quick iteration, that $F^\#(S^\#) \subseteq S^\#$ (*i.e.* that $S^\#$ is indeed a post-fixpoint of $F^\#$) and that $S^\# \sqsubseteq \phi$ (*i.e.* the certificate proves property ϕ).

More Formally

More formally, LANDE's PCC system uses the polyhedral domain *Poly* to express and prove liner relations between the program's variables. This domain does not, however, verify the ascending chain condition, so the analyzer computes a post-fixpoint $S^\#$ of the abstract semantics $[\![\mathbb{P}]\!]^\#$. Since the abstract semantics is the least fixpoint of the equation system (and also the least post-fixpoint), $S^\#$ is clearly a sound approximation of $[\![\mathbb{P}]\!]^\#$.
$S^\#$ is then pruned, in the hope that pushing it up in the lattice will allow for a smaller representation of it, and thus some compression [BJT07]. This pruning basically computes a witness $w \sqsupseteq S^\#$ such that $w \sqsubseteq \phi$ and $F(w) \sqsubseteq w$, the goal being to reach the most abstract of such witnesses.[2]
To make the checker more lightweight and easier to certify, it does not contain the full specification of the analyzer's abstract interpretation. Indeed, abstract unions on polyhedral domains are hard to compute (it is a convex hull computation) but cheap to verify (it is an inclusion check). The witness is then once again weakened by specifying invariants that should be verified at join points more vaguely, to ensure the inclusion check will be successful.
All these invariant weakening steps show that the analyzer computes more information than what is needed to perform the proof. This is, for now, a waste of ressources, but this extra information could be put to good use.
The checker itself is really simple: it abstracts $F^\#$ from $\mathbb{P}$ and performs all the necessary checks after having computed $F^\#(w)$. This simplicity, the fact that it is at the same time the most sensitive piece of the PCC puzzle, and the use of a formal framework such as abstract interpretation to specify it make it an excellent target for software formal certification. In this approach, it is certified in Coq.

2.2 Obfuscation

Obfuscation, however, is much less formal from the beginning. It has been used informally in the industry for years. The rise of bytecode-distributed languages made it necessary to study and compare the different existing obfuscation techniques. But the

[2]The reality of Turpin *et al.*'s algorithm is more gruesome, but also more efficient: it does not require w itself to verify the property, but rather $F^\#(w)$. This allows the final w to be even more abstract, even though some experiments show that being more abstract does not necessarily mean having a smaller representation.

lack of common formal framework made this work difficult. Collberg *et al.* [CTL97] make an extensive survey of different techniques that are commonly used on Java bytecode, and of different metrics used to assess their *resilience* to attacks. This does not, however, fit very well in the wanted formal background. We will here first follow Collberg *et al.*'s steps and make a quicker survey of existing obfuscation techniques before presenting recent work characterizing obfuscation using an abstract interpretation framework.

2.2.1 Syntactic Obfuscation

The survey by Collberg *et al.* describes two main classes of obfuscating transformations:

Data-flow obfuscation These obfuscation techniques rely on insertion of semantic `nop` instructions, or on replacing some sequences of instructions with semantically equivalent ones. They can be summed up in a few words as transformations that change the representation of the data in memory (that is, they change the execution trace of the program, the sequence of memory states it goes through).

Control-flow obfuscation These obfuscation techniques rely on in-depth modifications of the control-flow graph. Inserting non-reducible subgraphs (graphs that cannot be expressed in the source language, thus making decompilation difficult), for example is one of those modifications. The use of *opaque predicates* to insert conditional jumps is well-documented, as one of the most resilient obfuscation techniques.

In this syntactic, informal approach, an *opaque predicate* is a boolean expression whose value is fixed everytime it is reached, but is not known statically, except during compilation. Using this compile-time knowledge, arbitrary code can be inserted in the unreachable branch, making static analysis less precise, with the goal of making it unable to break the data-flow obfuscations that can be applied on the side.
However, this presentation is far too informal to provide a good comparison framework between obfuscation techniques. Different metrics are used to assess the "complexification" of the obfuscated program, that are based on various criteria that have more or less real value (loop nesting depth, number of variables, number of distinct variables, amount of dead code...).

2.2.2 Semantic Obfuscation

But obfuscation can also be seen more formally from a semantic point of view [DPG05b]. This novel approach by Dalla Preda *et al.*, based on abstract interpretation is quite easy to understand on data-flow obfuscation using Cousot's lattice of abstractions [Cou97]. Assume that you want to obfuscate a given semantic property (say, the parity of variable x). Then it is easy to define the most abstract abstraction that can remove this obfuscation. It is then even easier to compare two attackers with respect to a given obfuscated property, or two obfuscations (given by their obfuscated property) with respect to a given attacker.

Control-Code Obfuscation and Opaque Predicates

But the most interesting part of her work is done on the more efficient class of opaque predicate-based obfuscations [DPG05a]. Using abstract interpretation, and given an opaque predicate, they describe a generic method, based on domain completion with regard to an abstract operator,[3] to detect high-level opaque predicates, thus breaking the hardest part of the obfuscation [DPMDBG06]. Their work, however, is based on the assumption that they are the deobfuscator, since the goal is to develop a semantic-based malware detection method. This difference in perspective changes everything: when they only need sufficient conditions for deobfuscation, we want necessary characterizations of the abstractions that can break an opaque predicate; when they are concerned with completeness of the detection, we are more concerned with its soundness. But the ideas and formal framework Dalla Preda describes in her thesis were a good starting point to start looking for our own solutions. We will use her characterization of obfuscation using the lattice of abstractions, but will generalise some of the concepts and definitions she introduces.

[3] This is why the method is not an algorithm: domain completion is not decidable.

Formalizing the Problem

Having done this quick state of the art on PCC and obfuscation as we will use them, we can now make the problem at hand a little more formal, and precise where the important issues are.

Approach

It should be clear by now that the problem can be approached in many different ways, in that we can choose the order in which the operations are performed:

- obfuscate the program, then perform the analysis
- or, perform the analysis, then apply obfuscation.

Both these approaches have drawbacks. The first forces us to redesign the static analysis to work on obfuscated programs, which is counter-intuitive, since the goal of obfuscation is to render static analysis useless. We did not study this approach. The second one forces us to be careful not to lose the proof's soundness when obfuscating the program. This can be achieved by

- transforming the proof as we obfuscate the code,
- carefully choosing obfuscations to keep the proof sound as we go.

In this first approach of a solution, we chose to focus on the second method, trying to use the proof certificate as a guide to how we could safely obfuscate the program. We will, however, end up modifying certain parts of the certificates, thus getting a bit closer to the first proposition, without full-on modifying the proof.

Obfuscation Parameters

Now that we stated our goal to design an obfuscation technique that would integrate with Besson *et al.*'s PCC technique, we can study more precisely what parameters we have control on in order to perform this transformation. Remember that we will only consider opaque predicate-based obfuscations, since they are the only ones that allow good protection while staying at a rather high level of abstraction. We thus have some control over:

- the opaque predicates that we use
- and, the inserted (unreachable) code.

The first point is important, although classic, since it allows us to freely build our own opaque predicates. We will see that it is an important strength of the proposed technique, that allows for much better protection. The second point, is also important, and a bit more complicated to explain. Having control over the inserted code in fact gives control over the invariants that we insert or modify due to the presence of that unreachable code, in turn giving us some control over the information we end up giving an attacker.

The Attacker Model

Following Dalla Preda's lead, we will model our attacker as an abstract interpretation, with its abstraction ρ, and its abstract operators f^i_ρ. We will later assume it is able to perform basic operations on interpretations, such as reduced-products and complementations, and sometimes assume our attacker has access to the checker's abstraction. However, we will assume it is only able to perform operations that can be done automatically.

Chapter 3

A Semantic Solution

As we explained in section 2.2, Dalla Preda's work on semantic obfuscation, and especially her vision of opaque predicate detection, are not entirely adapted to our problem. Apart from all the differences listed there, it also appears that her element-wise approach does not correspond to the collecting semantics (see section 1.2) approach ordinarily used in static analysis. As a consequence, we will give here a new semantic definition of what an opaque predicate is, that also takes into account our position as the obfuscator (and thus want them to be harder to deobfuscate) –unlike Dalla Preda, who is in the deobfuscating position. Then we will give a definition of what detecting a predicate is opaque means in this framework, before showing how we can design and use such opaque predicates easily in conjunction with LANDE's PCC solution.

3.1 Semantic Control-Flow Obfuscation

The opaque predicates we want to define are a generalization of Dalla Preda's opaque predicates. Indeed, when she wants an opaque predicate to be either a tautology, or unsatisfiable, we only require it to be true *on the set of states that are reachable at the program point where it is inserted*. This makes them much harder to detect, since it does not only require an analysis of the predicate itself and the way it is built to determine its satisfiability (see [Dal07] and section 2.2), but also a forward analysis to determine the set of reachable states. Since all static analyses can provide is over-approximations of such sets, the attacking analysis needs to be adapted to the given program *and* the given predicate, to be able to compute accurately both the current subset of states X

and the satisfiability of the predicate on X. Moreover, we want this definition of opaque predicates to be given in a collecting semantics-like formalism.

Definition 3 *Opaque Predicate*
Let $\mathbb{P}$ be a program, and P be a predicate present at some program point l of $\mathbb{P}$. $\langle P, l\rangle$ is said to be an opaque predicate *whenever $[\![P]\!](X_l) = \emptyset$, where X_l is the set of reachable states at program point l.*[1][2]

It is necessary, in our definition, to formalize opaque predicate in a program-dependant fashion, since our approach in obfuscation will be greatly program-dependant itself.

3.1.1 Semantic Detection of Opaque Predicates

Detecting an opaque predicate is thus equivalent to computing that its output *on the given program* is always true (resp. false). However, this cannot generally be done, since program semantics are not, in general, computable. As a consequence, a deobfuscator will need to compute an abstraction of the program's semantic to try and detect opaque predicates. The definitions relevant to the semantic detection of opaque predicates follow.

Definition 4 *Concrete test*
We will call concrete test *associated to a predicate $\langle P, l\rangle$ in program $\mathbb{P}$ the test*

$$CT_{\mathbb{P}}(P, l) := [\![P]\!](X_l) = \emptyset \tag{3.1}$$

By definition, $CT_{\mathbb{P}}(P, l)$ if and only if P is opaque in $\mathbb{P}$

Thus, we will say that one detects an opaque predicate P in a program $\mathbb{P}$ whenever they find that $CT_{\mathbb{P}}(P, l)$ is true. However, as we mentioned before, this will not be, in general, directly verifiable. For an attacker computing on an abstract domain $\mathcal{D}^\rho$ characterized by the closure operator ρ, we define the following abstract test.

Definition 5 *Abstract Test*
We call abstract test *associated to a predicate $\langle P, l\rangle$ in program $\mathbb{P}$ with regard to an*

[1] X_l is the partial solution of the fixed-point equation, and $[\![P]\!]$ is the predicate transformer corresponding to P, as defined in appendix A.
[2] Note that all our opaque predicates are *false* on the reachable states. One can easily build partial tautologies (*true* on the reachable states) from a *false* opaque predicate by negating it.

abstraction ρ the test

$$AT^{\rho}_{\mathbb{P}}(P,l) := [\![P]\!]^{\rho}(X^{\rho}_{l}) = \bot_{\rho} \quad (3.2)$$

where X^{ρ}_{l} is the set of abstract states computed by the abstract program.[3]

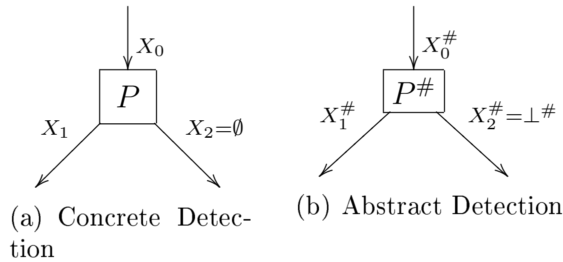
We define soundness and completeness of the abstract test (and thus of the corresponding abstraction) with regard to a given program and predicate in the same fashion as Dalla Preda.

Definition 6 *Abstract Test: Soundness and Completeness*
Let $CT_{\mathbb{P}}$ be a concrete test. We say that the asbtract test $AT^{\rho}_{\mathbb{P}}$ is:

- complete *with regard to $CT_{\mathbb{P}}$ when $CT_{\mathbb{P}} \Rightarrow AT^{\rho}_{\mathbb{P}}$.*
- sound *with regard to $CT_{\mathbb{P}}$ when $AT^{\rho}_{\mathbb{P}} \Rightarrow CT_{\mathbb{P}}$.*

Note that, for a given program and a given (set of) predicate(s), completeness and soundness depend only on the abstraction ρ. We will thus speak about completeness and soundness of abstractions –or domains– as well.

More intuitively, the abstract test is complete when it detects all opaque predicates, and it is sound when all the predicates it points out are indeed opaque.



(a) Concrete Detection (b) Abstract Detection

Figure 3.1: Concrete vs Abstract Detection of Opaque Predicates.

3.1.2 Soundness of the Abstract Test

Since we suppose our attacker works with a sound abstraction ρ, the abstract set of reachable states it computes on the inserted branch is an over-approximation of the concrete set of reachable states on that branch. Thus, finding it to be empty (*i.e.*

[3] That is, an over-approximation of $\rho(X_l)$, and not necessarily $\rho(X_l)$ itself.

detecting the opaque predicate) can only be done when the predicate is indeed opaque. Figure 3.1 shows a quick example of an opaque predicate, seen from the concrete and abstract point of view. This quick reasoning yields the following theorem:

Theorem 3 *Soundness of the Abstract Test Let ρ be a sound abstraction, then the abstract test $AT^{\rho}_{\mathbb{P}}$ is sound with regard to $CT_{\mathbb{P}}$.*

3.1.3 Completeness of the Abstract Test

Unlike Dalla Preda, we will not discuss the possibility for the attacker to improve its own domain to make it complete with regard to predicate detection on the attacked program. Indeed, we are under the assumption that the attacker is fully automated, and completeness refinements and extensions are not computable on general predicates. Moreover, unlike a signature-based malware detector, a reverse-engineer would care more about the soundness of his transformation (*i.e.* all removed predicates are indeed opaque) than about its completeness.

3.2 Preliminary Study: Opaque Predicates

Now that we have a clean definition of what an opaque predicate is, as well as a better understanding of our formal framework, we can study a bit more closely how obfuscation and proof certificates will need to interact. This will lead us to formulate new hypotheses regarding our obfuscator and the attacker model, that we will try prove realistic in chapter 4.

3.2.1 Opaque Predicates and Invariants

The first form of interaction between the proof certificate and deobfuscation is the direct transmission of semantic information in the form of invariants. Indeed, we have seen that the transmitted invariants can bee seen by the analysis as an `assert` statement, effectively filtering down the set of reachable states in a given branch. The concrete collecting semantic of an `assert(`*condition*`)` statement is a greatest lower bound (see appendix A) that can, if it yields $\perp$, prove the opacity of a predicate in a single and simple operation. The example in figure 3.2 shows how a bad choice of invariants can make

deobfuscation trivial. Clearly, since the invariants that are computed on the original program are sound, they will not cause any trouble. However, inserting a predicate that is not "compatible" with them, or inserting code in the dead branch that requires the invariants to falsify the invariant chain will make deobfuscation as easy as computing an intersection. This leads us to formulate a first rule regarding our system:

> We will make sure that our obfuscator is "intelligent", in that it chooses inserted invariants and predicates so that they do not trivially become false on their direct input.

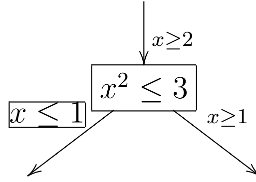


Figure 3.2: An example of badly chosen invariants (the arrows hold the invariants)

Clearly, having control on the bits of code that are inserted in the dead branches, as well as on the opaque predicates themselves, makes this assumption entirely realistic, since we can build arbitrary code that keeps any invariant valid.

3.2.2 Opaque Predicates and Invariant Composition

It should be clear, regarding the semantics of `assert` instructions, that any finite number of invariants compose and commute the way you would expect them to do. Indeed, if I_1 and I_2 are invariants (or properties, or sets of environments...) and $[\![I_1]\!]^{\downarrow}$, $[\![I_2]\!]^{\downarrow}$ the respective corresponding collecting semantic transformers. Recall the definition given in section A.2.2:

$$[\![I]\!]^{\downarrow}(X) = \{\sigma \in X | \sigma \in I\}$$

Thus, we have the following:

$$[\![I_2]\!]^{\downarrow} \circ [\![I_1]\!]^{\downarrow}(X) = \{\sigma \in \{\sigma \in X | \sigma \in I_1\} | \sigma \in I_2\}$$

And, by a quick development,

$$[\![I_2]\!]^{\downarrow} \circ [\![I_1]\!]^{\downarrow}(X) = \{\sigma \in X | \sigma \in I_1 \wedge \sigma \in I_2\}$$

We can deduce and conclude:

Theorem 4 *Concrete invariants composition*
In the concrete semantics we defined for our `While` *language, we have that, for all properties I_1 and I_2:*

$$[\![I_1]\!]^{\downarrow} \circ [\![I_2]\!]^{\downarrow} = [\![I_1 \cap I_2]\!]^{\downarrow} = [\![I_2]\!]^{\downarrow} \circ [\![I_1]\!]^{\downarrow}$$

However, this is not true anymore in general in the abstract world, since the abstract greatest lower bound $\bigsqcap$ may not have the same commutation and associativity properties as its concrete counterpart, which is basic set intersection. Since we will want to reason on compositions of backtests (for example, to check that our invariants verify the previous hypothesis), we will make the following assumption:

> We suppose that the attackers abstract conjunctions of concrete properties as conjunctions of abstract properties (*i.e.* $[\![I_1 \wedge I_2]\!]^{\downarrow\rho} = [\![I_1]\!]^{\downarrow\rho} \sqcap_{\rho} [\![I_2]\!]^{\downarrow\rho}$)

This assumption is realistic as soon as the abstract domain is complicated enough. For simpler domains (for example the interval domain), it is easy to find abstract predicate transformers that perform much better in terms of precision than the ones defined as in our assumption. We will see, however, that in order to avoid opaque predicate detection, we will need to make them detectable only by complicated domains, and the assumption is then reduced to an assumption concerning the complexity of the abstraction required to detect the predicate.

3.3 A Solution

Having made those first assumptions, we can now give the details of our obfuscation technique. Remember our main goals here are to produce a reasonably efficient (in terms of obfuscating power) obfuscation technique that is not made easier to reverse by the presence of sound invariants, and that does not prevent the checking of the proof invariants. To do this, we have to design predicates that are true on the set of reachable states at the program point where they are inserted, but are complicated enough to not be trivialy detected.

3.3.1 Building Opaque Predicates

The main idea to build these opaque predicates is to reuse the analysis that has to be performed for proof purposes. In fact, the PCC analyzer computes, for each program point, an over-approximation X_l^{PCC} of reachable states. Thus, any predicate that is true on X_l^{PCC} will also be true on the real set of reachable states. Moreover, as explained in section 2.1 all X_l^{PCC}'s are not transmitted along with the program and in fact need to be rebuilt by the checker in a simple iteration of the equation system. We can then use the pruned parts of the invariants themselves to build much more complicated opaque predicates, *i.e.* predicates that require a much more concrete analysis to be detected (we will take here the example of polynomial relationships between variables).
For example, if our analyzer computes an invariant $I = x \geq 2 \wedge x \leq y \geq 2x$ and only $I' = x \leq y \wedge x \geq 2$ is transmitted, then we can use the extra information yielded by the original invariant to build, for example, a polynomial predicate of arbitrary degree: $P = y^2 - 4x^2 - 3x \leq 0$. Of course, real invariants involve more than a simple couple of variables, and the relations between them can be complicated enough to require an advanced analysis domain to discover.

Making Invariant Rebuilding Difficult

However, one can argue that the invariants are pruned in such a way that they can be reconstructed easily by a simple iteration of the checker's abstract representation of the program, and that the checker, being on the client-side, is fully available to the deobfuscator. It is the case that the invariants can be *partly* reconstructed in such a way, but this method does not, in any case, allow to reconstruct the entire certificate, even on the original program, since this would require the attacker to know the full extent of the analyzer, widening heuristics included. Moreover, the program that is analyzed by the deobfuscator is the obfuscated version, where the control-flow graph is scrambled, and the invariant reconstruction is thus even less complete, especially when the opaque predicate is not understood by the attacker.

> We will thus assume, in this first approach, that, if ρ denotes the attacker's abstraction and P denotes an opaque predicate, we always have $[\![P]\!]^{\downarrow\rho} = id$. Namely, the attacker is unable to extract any semantic information from the opaque predicate.

This hypothesis is, of course, not possible to ensure in the general case: it is always possible to design an analysis that will understand a given predicate. However, we could use, for example Dalla Preda's "interleaving" technique to prevent deobfuscation of one opaque predicate by making it interact with some other opaque predicate that requires a different analysis. Besides, we are currently working on trying to weaken this assumption by considering some domain theory results (see chapter 4). Finding such a weaker assumption could also solve the problem of the *stealth* of our predicates. Indeed, such predicates can easily be detected as the only ones who are not understood by the attacker's abstraction, since the other ones need to be at least understandable by the checker, that is available to the attacker. This issue will be addressed in section 4.2. Still, how can we be sure that this hypothesis prevents an attacker from detecting the predicate? Consider a program $\mathbb{P}$, with an opaque predicate P at program point l, on the one hand, and program $\mathbb{P}'$ that is simply $\mathbb{P}$ where P has been replaced at l by a non-deterministic branch selection (see appendix A). If the attacker's abstraction ρ is such that $[\![P]\!]^{\downarrow\rho} = id$, then the abstract views of $\mathbb{P}$ and $\mathbb{P}'$ are equal and the two programs are undifferentiable by the attacker. Clearly then, finding that P is opaque would be equivalent to finding that the corresponding non-deterministic branch is never taken. According to the classic non-deterministic semantics, that means that the branch-selection instruction itself is never executed. By pushing this non-reachability upwards in the control-flow graph, we reach either:

- a loop. That means that this entire loop is dead code and non-connex with the rest of the CFG, and as such, should not even have been analyzed.[4]
- a deterministic branching. That means that the corresponding predicate is opaque and the same proof applies recursively to the upper subgraph (remember we do not consider arbitrary CFGs here).
- the program's entry point. That means that the entire program can never be executed.

In all three cases, the proof ends by concluding that this situation is not possible if the program was originally clear of dead content (which anyway does not need obfuscated or protected).[5] Thus, an attacker that verifies our assumption can never, unless the

[4]If this loop was in fact reachable, then we reach either an opaque predicate and proceed with the recursion, or a non-opaque branching and proceed using the following item.

[5]And if it did need obfuscated for some reason (*e.g.* because the program is a patched version but the patched algorithm is still proprietary), such a deobfuscation process could not be considered sound on the piece of dead code, since it would remove it all.

invariants give it away and make the dead branch trivially non-executable, detect an opaque predicate.

Making our Predicates Truly Opaque

In the example given at the beginning of this section, we quickly described a way of building opaque predicates from linear relationships between variables. Ths given method, which consists in building a polynomial invariant I' from a linear invariant I, such that $I \Rightarrow I'$ can be very efficient despite its apparent simplicity. Indeed, there are very few polynomial decision heuristics, and this process is anyway very expensive.

Chapter 4

An Abstract Point of View

Throughout this chapter, and unless otherwise specified, we will confuse abstraction, domain, interpretation and closure operator, and we will always assume that an abstract interpretation is equipped with the best corresponding abstract operators.
We first present some original concepts that could lead to refining our assumptions with regard to the attacker. Indeed, we could in that way characetrize the power of an attacker that can perform reduced-products of abstractions. Then we will quickly present some of the leads that we follow to solve the stealth problem, and that could also help make our opaque predicates more resilient.

4.1 Partitionning the Lattice of Abstractions

In an attempt to formalize our hypothses in a more abstract manner, and especially to give a measure of what the deobfuscator should be allowed to perform, we tried to characterize more precisely two particular abstractions in the lattice of abstractions, given a program $\mathbb{P}$ and a set of opaque predicates OP. Let us first denote $\mathcal{B}_{\mathbb{P}}(OP)$ [resp. $\overline{\mathcal{B}_{\mathbb{P}}}(OP)$] the set of abstractions that detect some [resp. do not detect any] predicates in OP that are present in $\mathbb{P}$. These two sets clearly form a partition of the set of abstractions. We define $\rho^*_{\mathbb{P}}(OP) = \bigsqcup \mathcal{B}_{\mathbb{P}}(OP)$ (intuitively, it would be the most abstract domain that detects all predicates, if it was in $\mathcal{B}$) and $\overline{\rho^*_{\mathbb{P}}}(OP) = \bigsqcap \overline{\mathcal{B}_{\mathbb{P}}}(OP)$.

4.1.1 Monotonicity of Computational Power

Before going further, it is necessary to present a first, very intuitive result, namely what we will call the *monotonicity of computational power*. Simply put, this means that if an abstraction ρ allows one to compute some property P on a program $\mathbb{P}$, all abstractions that are more concrete than ρ (*i.e.* smaller than, in the lattice of abstractions) allow to compute P on $\mathbb{P}$ as well, by simply applying ρ on top of the computation. The main consequence of this for us is the following.

Theorem 5 *Monotonicity of Predicate Detection*
Let $\mathbb{P}$ and OP be a program and a set of opaque predicates, and let $\rho \in \mathcal{B}_{\mathbb{P}}(OP)$ be an abstraction that detects some predicates of OP in $\mathbb{P}$. Then, for all $\rho' \sqsubseteq \rho$, $\rho \in \mathcal{B}_{\mathbb{P}}(OP)$. The dual holds.

We speak here of monotonicity because we can clearly reformulate the theorem by considering, with $\mathbb{P}$ fixed, the least upper bound OP_{max} of the sets of predicates that an abstraction can detect, and state that the more precise the abstraction, the bigger OP_{max}.

4.1.2 Direct Results

We now would like to study case by case, what it means for $\rho^*_{\mathbb{P}}(OP)$ to be in $\mathcal{B}_{\mathbb{P}}$, and for $\overline{\rho^*_{\mathbb{P}}}(OP)$ to be in $\overline{\mathcal{B}_{\mathbb{P}}}$. As a direct result of theorem 5, we can eliminate all possibilities where $\overline{\rho^*_{\mathbb{P}}}(OP) \sqsupseteq \rho^*_{\mathbb{P}}(OP)$, since we have the following:

$$\forall \rho \in \mathcal{B}_{\mathbb{P}}(OP), \rho \sqsubseteq \rho^*_{\mathbb{P}}(OP)$$

and

$$\forall \rho \in \overline{\mathcal{B}_{\mathbb{P}}}(OP), \rho \sqsupseteq \overline{\rho^*_{\mathbb{P}}}(OP)$$

Then, either we have $\overline{\rho^*_{\mathbb{P}}}(OP) \in \overline{\mathcal{B}_{\mathbb{P}}}(OP)$ and $\rho^*_{\mathbb{P}}(OP) \in \mathcal{B}_{\mathbb{P}}(OP)$, which would lead to a lattice with the general hourglass shape given in figure 4.1(a), which seems impossible to get for a lattice of abstractions; or we have $\overline{\rho^*_{\mathbb{P}}}(OP) = \rho^*_{\mathbb{P}}(OP)$, which would make them both either $\top$ or $\bot$[1], making the obfuscation impossible to undo (which is not possible), or trivially breakable (which is not wanted) respectively.
We can also deal easily with the case where $\rho \in \overline{\mathcal{B}_{\mathbb{P}}}(OP)$ and $\overline{\mathcal{B}_{\mathbb{P}}}(OP)$ are not comparable. In that case, we necessarily have $\overline{\rho^*_{\mathbb{P}}}(OP) \in \overline{\mathcal{B}_{\mathbb{P}}}(OP)$ and $\rho^*_{\mathbb{P}}(OP) \in \mathcal{B}_{\mathbb{P}}(OP)$

[1] Because it cannot both break and not break predicates in OP and because of theorem 5

(otherwise the two ρ^*s would be comparable), and this once again leads to a strange-shaped lattice, shown in figure 4.1(b), that is not intuitively compatible with being a lattice of abstractions.
The interesting cases are the ones where $\overline{\rho^*_{\mathbb{P}}}(OP) \sqsubset \rho^*_{\mathbb{P}}(OP)$. We can distinguish three different setups:

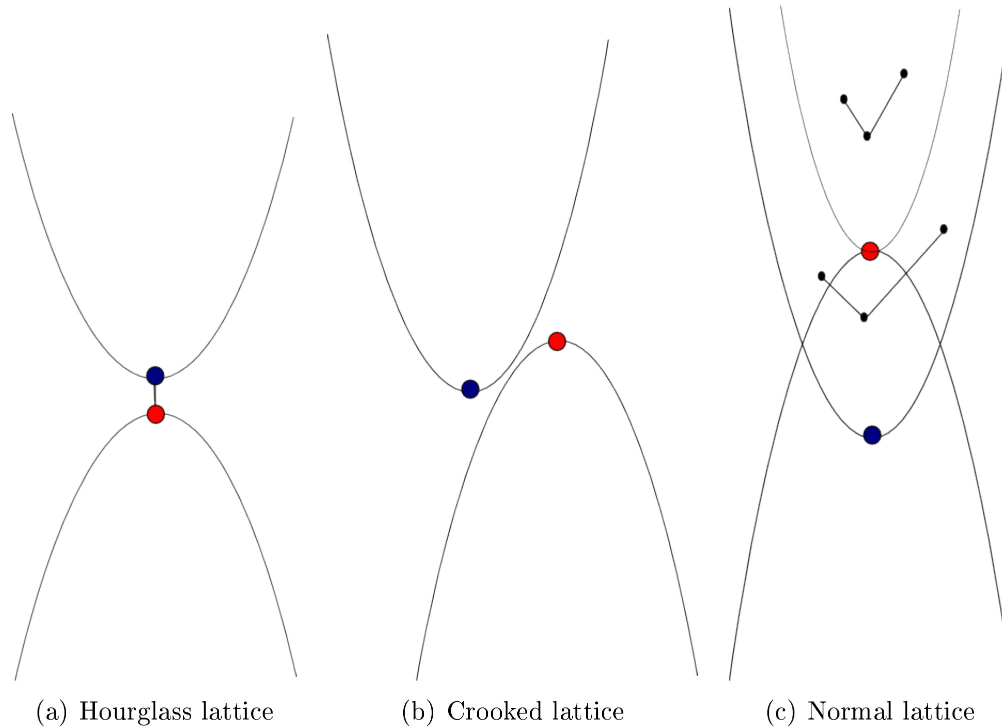
$\overline{\rho^*_{\mathbb{P}}}(OP) \in \overline{\mathcal{B}_{\mathbb{P}}}(OP)$. In that case, any reduced product between two abstractions that do not detect any of the predicates in OP will still belong to $\overline{\mathcal{B}_{\mathbb{P}}}(OP)$. This would be the best case for us, since it would allow to consider a wider class of attacker, with some amount of human reasoning aloowing them to perform reduced products *and* get the best abstract operators for the resulting abstractions. Moreover, the condition that our checker is more abstract than $\overline{\rho^*_{\mathbb{P}}}(OP)$ could be checked in a sound (but incomplete) manner even when we cannot describe $\overline{\rho^*_{\mathbb{P}}}(OP)$ itself, by comparing it with some known abstraction that does not detect any predicates –which would be an over-approximation of $\overline{\rho^*_{\mathbb{P}}}(OP)$.

$\rho^*_{\mathbb{P}}(OP) \in \overline{\mathcal{B}_{\mathbb{P}}}(OP)$ and $\overline{\rho^*_{\mathbb{P}}}(OP) \notin \overline{\mathcal{B}_{\mathbb{P}}}(OP)$. In that case, we will require all considered abstractions ρ to be such that $\rho \sqsupset \rho^*_{\mathbb{P}}(OP)$ to be able to ensure that their reduced product does not detect any of the predicates. However, we will have no way of checking whether or not that condition is verified without being able to compute a sound over-approximation of $\rho^*_{\mathbb{P}}(OP)$, which is not doable for the moment.

$\rho^*_{\mathbb{P}}(OP) \notin \overline{\mathcal{B}_{\mathbb{P}}}(OP)$ and $\overline{\rho^*_{\mathbb{P}}}(OP) \notin \overline{\mathcal{B}_{\mathbb{P}}}(OP)$. In this case, the "zone" between $\rho^*_{\mathbb{P}}(OP)$ and $\overline{\rho^*_{\mathbb{P}}}(OP)$ will play the role of a fuzzy border between the two partitions of the lattice. As a consequence, and even if the global conditions to ensure that a reduced-product-enabled attacker does not detect predicates are the same as in the previous case, we can hope to weaken them if we manage to characterize $\overline{\mathcal{B}_{\mathbb{P}}}(OP)$ and $\mathcal{B}_{\mathbb{P}}(OP)$

4.1.3 Open Leads

Given more time, we could explore, given a certain number of programs and some classes of opaque predicates, if some of them are in the good cases, where we can theoretically ensure that even an attacker that can combine abstractions will not be helped by any bit of knowledge from the checker. Moreover, simply knowing in what cases $\rho^*_{\mathbb{P}}(OP)$ does break some predicates would give much better confidence in the fact that Dalla



(a) Hourglass lattice (b) Crooked lattice (c) Normal lattice

Figure 4.1: Shapes of Abstraction Lattices

Preda's work on completeness can be adapted to our search for necessary conditions for deobfuscation.

4.2 Making our Opaque Predicates Stealthy

As it was quickly stated at then end of the previous chapter, our technique does not, so far, produce *stealthy* predicates –that is predicates that can be marked as "suspicious" by the attacker without any advanced analysis. Indeed, the original predicates need to be understood by the checker so the invariants can be rebuilt and refined enough to prove the wanted safety properties. Yet, we supposed that our opaque predicates are such that they do not give any sort of semantic information to the attacker. Thus, chances are they will be the only ones the deobfuscator won't be able to understand at first. We can, however, solve this problem in different ways.

4.2.1 Masking Standard Predicates

The first approach we could use is to make normal predicates look as if they were opaque to the deobfuscator. Since we assumed soundness was a concern, he will then not be able to use the fact that, for a predicate $\langle P, l\rangle$, $[\![P]\!]^{\downarrow\rho}(X_l) = \bot_\rho$ to mark $\langle P, l\rangle$ as opaque. We can try to achieve this by identifying which of the original predicates need to be understood by the checker to reconstruct a precise enough invariant to perform the proof and soundly make them look opaque, for example by taking their disjunction with a *false* opaque predicate built using the original invariant. The difficulty there lies in identifying which predicates can be masked in such a way –and to what extent– and which need to be preserved for the proof-checking to go as planned.

4.2.2 Unmasking Opaque Predicates

The second solution to this stealth problem would be to allow the deobfuscator to get some information out of opaque predicates, but to make sure that information was useless for deobfuscating purposes. A quick intuition of this idea would be if all invariants were based on relationships between variables x and y, and the analysis of the predicate gave information about some independent variable z. Of course, such a predicate would not be stealthy either and the technique needs to be refined.

Using Domain Theory

To achieve this goal, we could for example use Cortesi *et al.*'s works in domain theory and theory of abstract interpretations [CFW92, CFW94, CFG+95].

Complementation Complementation would, for example, allow us to compute the abstract domain ρ that is such that $\rho \sqcup Poly = \mathcal{D}$ and $\rho \sqcap Poly = id$, in order to try and design opaque predicates that let information leak on domain ρ and domains "close to" ρ, so that the leaked information is useless in order to reconstruct polyhedral invariants. Depending on the results of such a complementation, such opaque predicates could be extremely stealthy *and* hard to detect using standard domains.

Comparing Abstract Interpretations On the other hand, quotients of abstract interpretations allow to compare interpretations (as opposed to comparing simple abstrac-

tions, which is allowed by Cousot's hierarchy of abstractions) in a very precise manner, *with regard to a reference property.* Such theoretical tools could help characterize $\rho^*_{\mathbb{P}}(\{\langle P, l\rangle\})$ for some predicate $\langle P, l\rangle$, as well as the most concrete domain that does not allow to reconstruct the original invariant, and we could allow our predicates to leak information only in domains that are more abstract than their *lub*.

Conclusion

In this report, after briefly introducing the concepts of abstract interpretation and reviewing the state of the art in proof-carrying code and obfuscation, we presented a new characterization of control-flow obfuscation using opaque predicates. This formalization allows for powerful control-flow obfuscations that are however capable of preserving the soundness of a proof certificate, as well as its completeness with regard to a given safety property. We then presented new results in domain theory, related to deobfuscation and detection of opaque predicates.

Future Work

By its very nature as a first exploration of a field, this project raised a number of questions. Possible extensions of it could lead to a formalization of the stealth of an obfuscating transformation that would improve the resilience of the presented method. Future work could also explore the ideas presented on the lattice of abstractions to define classes of programs and predicates that allow us to ensure better properties on our obfuscated programs, but that could also enlighten us on what can and what cannot be obfuscated. Finally, there is room for much improvement using domain theoretical results, especially the works that take into account the intepretation as well as the abstraction, since they allow for a finer, more precise comparison of the computational power of two attackers with regard to a given obfuscation (or, on the other hand, allow to express the needed differences between the checker and the attacker model).
An important step forward in the study of this problem would be to implement a prototype of the defined obfuscation technique to perform tests. This could also help finding exampls for each one of the cases highlighted in chapter 4.

Acknowledgements

I would like to thank David Pichardie for allowing me to further the development of my formal background, introducing me to abstract interpretation, making me realize software security was not only hand-waving and coming up with such a great topic. I regret I was not able to go further in so little time and I regret I had to choose not to keep working full-time on this project. I hope this report helps whoever comes next.
I would like to thank the anonymous referee who pointed out the *amanat* cryptographic solution existed, bringing a little more completeness to this report.
I would like to thank the whole of the LANDE team for the great ideas they came up with, the support they gave me, and for making me realize the semantic study of obfuscation could have consequences in other fields as well.
I would like to thank Luc Bougé, Robin Cockett, Daniel Hirschkoff, and David Pichardie for their huge help in my quest for an English speaking PhD position. Thanks also go to David Aspinall, Ian Stark and John Longley in Edinburgh, and Andrew Gordon and Cédric Fournet in Cambridge for their time and patience when I was tediously trying to explain how abstract interpretation could help them. Thanks to Jan Jürjens and the Open University Research School for their open doors and final decision.
Finally, I would like to thank those who supported me, especially Loïc and the drinks at Tiffany's, and mostly Heather and her discreet, distant but ever present love.

Appendix A

A `While` language

In this section, we quickly define and present the `While` language, as well as a variant that will be used in a proof. We first give its syntax before giving the definitions of some of its semantics: structural operational, big-step semantic, and collecting semantic.

A.1 Syntax

Figure A.1 shows the basic syntax of our `While` language. We define it to be very simple, without loss of generality, since it is as expressive as any other. However, we have to note that this syntax is closer to a source language than it is to any bytecode or intermediate language. We chose this approach because our goals are rather high-level and human reasoning on the source language is much easier than on a machine-readable form.
More important to our problem, you will remark that the restrictions concerning conditional and unconditional jumps –operations that form the core of the specific obfuscation technique we will study in this report– do not correspond to Java bytecode. This is due to the fact that we assume our system works on the output of a non-optimizing compiler. Although this also restricts our obfuscation power, as we cannot introduce non-reducible subgraphs (see section 2.2 for more on obfuscation techniques), it should not make the reasoning any less sound, since our goal is here to prove that we provide an efficient technique. Thus, by restricting its power, we would only risk making it less efficient.

A.1.1 Common Shorthands

The toy example shown in figure A.2(a) will be used throughout the report and shows different syntactic shorthands that we will also use repeatedly. Figure A.2(b) shows the equivalent program using the pure syntax described in figure A.1.

$$
\begin{array}{llll}
Exp & ::= & n & n \in \mathbb{Z} \\
 & | & ? & \\
 & | & x & x \in \mathbb{V} \\
 & | & Exp\ o\ Exp & o \in \{+, -, \times\} \\
 & & & \\
test & ::= & Exp\ c\ Exp & c \in \{=, \neq, <, \leq\} \\
 & | & test\ \texttt{and}\ test & \\
 & | & test\ \texttt{or}\ test & \\
 & & & \\
Stm & ::= & {}^{l}[x\ :=\ Exp] & l \in \mathbb{L} \\
 & | & {}^{l}[\texttt{skip}] & \\
 & | & \texttt{if}\ {}^{l}[test]\ \texttt{then}\{\ Stm\ \}\{\texttt{else}\ Stm\ \} & \\
 & | & \texttt{while}\ {}^{l}[test]\ \{\ Stm\ \} & \\
 & | & Stm\ ;\ Stm & \\
 & & & \\
Prog & ::= & [Stm]^{end} &
\end{array}
$$

where $\mathbb{L}$ is the set of labels for program points (we will usually use $\mathbb{N}$), and $\mathbb{V}$ is the set of variable names.

Figure A.1: The `While` syntax.

A.2 Semantics

The semantics are very basic. The environment Env is a mapping from variable names to their value: $Env = \mathbb{V} \rightarrow \mathbb{Z}$, and the states are pairs of a program point and an environment: $States = \mathbb{L} \times Env$. Figures A.3(a) and A.3(b) show the arithmetic expression and boolean expression semantics, respectively.

A.2.1 Control-Flow Graph

The control-flow graph (CFG) is a standard representation of programs for static analysis. We give here a quick definition of the graph model and its semantics.

```
1:  x:=0;
2:  y:=0;
3:  while (x<6) {
4:     if (?<?)  {
5:        y:=y+2;
       };
6:     x:=x+1;
    }
```

(a) A toy program

```
[¹[x:=0;]
²[y:=0;]
while ³[x<6] {
   if ⁴[?<?]  then{
      ⁵[y:=y+2;]
   } else {
      ⁵'[skip;]
   };
   ⁶[x:=x+1;]
};]^end
```

(b) The same "pure" program

Figure A.2: A syntax for `While`.

Graph Model

- The nodes are program points $k \in \mathbb{L}$.
- The edges are labeled with basic instructions:

$Instr$	$::=$	$x := Exp$	assignment
	$\mid$	**assert** $test$	test

- The CFG is a triplet (k_{init}, S, k_{end}) with
  - $k_{init} \in \mathbb{L}$ the entry point,
  - $k_{end} \in \mathbb{L}$ the exit point,
  - $S \subseteq \mathbb{L} \times Instr \times \mathbb{L}$ the set of edges.

Small-step semantics

- We first define the semantics of instructions: $\xrightarrow{i} \subseteq Env \times Env$

$s \rightarrow^{x:=a} s[x \mapsto v]$	for all $v \in \mathcal{A}[\![a]\!]_s$
$s \rightarrow^{\texttt{assert}\,t} s$	if $\mathbf{tt} \in \mathcal{B}[\![t]\!]_s$

$$\mathcal{A}[\![n]\!]_\rho = \{n\}$$
$$\mathcal{A}[\![?]\!]_\rho = \mathcal{Z}$$
$$\mathcal{A}[\![x]\!]_\rho = \{\rho(x)\}, x \in \mathbb{V}$$
$$\mathcal{A}[\![e_1\ o\ e_2]\!]_\rho = \{v_1\ o\ v_2 \mid v_1 \in \mathcal{A}[\![e_1]\!]_\rho, v_2 \in \mathcal{A}[\![e_2]\!]_\rho\}$$

(a) Semantic of arithmetic expressions

$$\frac{v_1 \in \mathcal{A}[\![e_1]\!]_\rho \quad v_2 \in \mathcal{A}[\![e_2]\!]_\rho \quad v_1\ \bar{c}\ v_2}{\mathbf{tt} \in \mathcal{B}[\![e_1\ c\ e_2]\!]_\rho}$$

$$\frac{v_1 \in \mathcal{A}[\![e_1]\!]_\rho \quad v_2 \in \mathcal{A}[\![e_2]\!]_\rho \quad \neg(v_1\ \bar{c}\ v_2)}{\mathbf{ff} \in \mathcal{B}[\![e_1\ c\ e_2]\!]_\rho}$$

$$\frac{b_1 \in \mathcal{B}[\![t_1]\!]_\rho \quad b_2 \in \mathcal{B}[\![t_2]\!]_\rho}{b_1 \wedge_{\mathbb{B}} b_2 \in \mathcal{B}[\![t_1\ \mathbf{and}\ t_2]\!]_\rho}$$

$$\frac{b_1 \in \mathcal{B}[\![t_1]\!]_\rho}{b_1 \vee_{\mathbb{B}} b_2 \in \mathcal{B}[\![t_1\ \mathbf{and}\ t_2]\!]_\rho} \qquad \frac{b_2 \in \mathcal{B}[\![t_2]\!]_\rho}{b_1 \vee_{\mathbb{B}} b_2 \in \mathcal{B}[\![t_1\ \mathbf{and}\ t_2]\!]_\rho}$$

(b) Semantic of boolean expressions

with $e \in Exp, t \in test, \rho \in Env$ and $\mathcal{B} = \{\mathbf{tt},\ \mathbf{ff}\}$.

Figure A.3: Semantics for `While` expressions.

- Then a small-step relation $\rightarrow_p \subseteq State \times State$ for a CFG $p = (k_{init}, S, k_{end})$

$$\frac{(k_1, i, k_2) \in S \quad s_1 \xrightarrow{i} s_2}{(k_1, s_1) \rightarrow_p (k_2, s_2)}$$

- The set of initial states is $\mathcal{S}_0 = \{k_{init}\} \times Env$
- And the set of reachable states is $[\![p]\!] = \{s \mid \exists s_0 \in \mathcal{S}_0, s_0 \rightarrow_p^* s\}$

A.2.2 Collecting Semantic

The collecting semantics defines the set of reachable states $[\![\mathbb{P}]\!]_k^{col}$ for each program point k of program $\mathbb{P}$. We thus define

$$\forall k \in \mathbb{L}, [\![\mathbb{P}]\!]_k^{col} = \{\sigma \mid (k, \sigma) \in [\![\mathbb{P}]\!]\}$$

where $[\![\mathbb{P}]\!]$ is the set of reachable states defined on the corresponding CFG, as defined in the previous paragraph.
It can be proved that $[\![\mathbb{P}]\!]^{col}$ is the least fixpoint of the following system:

$$\forall k \in \mathbb{L}, X_k = X_k^{init} \cup \bigcup_{(k',i,k)\in p} [\![i]\!](X_{k'})$$

with $X_k^{init} = Env$ if $k = k_{init}$ and $X_k^{init} = \emptyset$ otherwise, and $[\![i]\!] = X \mapsto \{s_2 | \exists s_1 \in X, s_1 \xrightarrow{i} s_2\}$.

A.3 Non-Deterministic Extension: ND-While

We will need, in one proof, to extend the language by adding a non-deterministic branching instruction to it. We will denote it `select {Stm}{Stm}`. Its semantic interpretation is equivalent to that of an `if` statement with a non-deterministic condition: if the condition is executed with set of reachable states X, then *both* branches are executed with sets of reachable states X.

Bibliography

[APH04] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. Abstraction-carrying code. In *Logic for Programming, Artificial Intelligence and Reasoning*, pages 380–397, 2004. 16

[BJPT07] Frédéric Besson, Thomas Jensen, David Pichardie, and Tiphaine Turpin. Result certification for relational program analysis. Rapport de recherche 6333, INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France, October 2007. 16

[BJT07] Frédéric Besson, Thomas P. Jensen, and Tiphaine Turpin. Small witnesses for abstract interpretation-based proofs. In *ESOP*, pages 268–283, 2007. 17

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. 10

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. 11

[CC04a] Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, January 14-16 2004. ACM Press, New York, NY. 6

[CC04b] Patrick Cousot and Radhia Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004. 11

[CFG+95] Agostino Cortesi, Gilberto Filé, Roberto Giacobazzi, Catuscia Palamidessi, and Francesco Ranzato. Complementation in abstract interpretation. In *SAS*, pages 100–117, 1995. 37

[CFW92] Agostino Cortesi, Gilberto Filé, and William H. Winsborough. Comparison of abstract interpretations. In *ICALP*, pages 521–532, 1992. 37

[CFW94] Agostino Cortesi, Gilberto Filé, and William H. Winsborough. The quotient of an abstract interpretation for comparing static analyses. In *GULP-PRODE (1)*, pages 372–387, 1994. 37

[Cou97] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. 13, 19

[CSV07] Sagar Chaki, Christian Schallhart, and Helmut Veith. Verification across intellectual property boundaries. In *Computer Aided Verification*, pages 82–94, 2007. 5

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science – The University of Auckland, July 1997. 4, 18

[Dal07] Mila Dalla Preda. *Code Obfuscation and Malware Detection by Abstract Interpretation*. PhD thesis, Università degli Studi di Verona – Dipartimento di Informatica, 2007. 6, 23

[DPG05a] Mila Dalla Preda and Roberto Giacobazzi. Control code obfuscation by abstract interpretation. In *SEFM*, pages 301–310, 2005. 19

[DPG05b] Mila Dalla Preda and Roberto Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *ICALP*, pages 1325–1336, 2005. 19

[DPMDBG06] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque predicates detection by abstract interpretation. In *AMAST*, pages 81–95, 2006. 19

[Nec97] George Necula. Proof-carrying code. In *POPL '97: Proceedings of the*

24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 106–119, New York, NY, USA, 1997. ACM. 5, 15