Obfuscation by Partial Evaluation of Distorted Interpreters

Roberto Giacobazzi
University of Verona
roberto.giacobazzi@univr.it

Neil D. Jones
DIKU, University of Copenhagen
neil@diku.dk

Isabella Mastroeni
University of Verona
isabella.mastroeni@univr.it

Abstract

How to construct a general *program obfuscator*? We present a novel approach to automatically generating obfuscated code P′ from any program P whose source code is given. Start with a (program-executing) interpreter `interp` for the language in which P is written. Then "distort" `interp` so it is still correct, but its specialization P′ w.r.t. P is transformed code that is equivalent to the original program, but harder to understand or analyze. Potency of the obfuscator is proved with respect to a general model of the attacker, modeled as an approximate (abstract) interpreter. A systematic approach to distortion is to make program P obscure by transforming it to P′ on which (abstract) interpretation is *incomplete*. Interpreter distortion can be done by making residual in the specialization process sufficiently many interpreter operations to defeat an attacker in extracting sensible information from transformed code. Our method is applied to: code flattening, data-type obfuscation, and opaque predicate insertion. The technique is language independent and can be exploited for designing obfuscating compilers.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program Synthesis; F.3.2 [*Semantics of Programming Languages*]: Partial Evaluation, Program Analysis

General Terms Languages, theory

Keywords Obfuscation, semantics, partial evaluation, program transformation, program interpretation, abstract interpretation.

1. Introduction

Code obfuscation is increasing its relevance in security, providing an effective way for facing the problem of both source code protection and binary protection. This contributes to comprehensive digital asset protection, with relevance to applications in DRM systems, IPP systems, tamper resistant applications, watermarking and fingerprinting (see [7] for a comprehensive survey on the literature), and white-box cryptography [5].

1.1 The problem

The idea of code obfuscation is simple and relies upon making security inseparable from code: *a program, or parts of it, are transformed in order to make them hard to understand or analyze* [8]. This may provide room for hiding cryptographic keys or sensible information concerning algorithmic design, making a direct attack to software in an untrusted environment harder, in a *security by obscurity* approach. Results on the impossibility of perfect and universal obfuscation, such as [2], did not dishearten researchers in developing methods and algorithms for hiding sensitive information in programs. By analogy with Rice's theorem (a great challenge for the development of automatic program analysis and verification tools), the impossibility of perfect obfuscation against malicious host attacks is a major challenge for developing concrete techniques that are sufficiently robust that an attacker is in trouble for an unacceptable amount of time in trying to defeat them. In digital contexts, hiding information means both hiding as in making imperceptible, as in watermarking, and obscuring as in making incomprehensible, as in obfuscation [30]. In programming languages, both aspects have deep semantic aspects: on one hand modified code should behave consistently with its source version yet hiding secret information, on the other hand, analysis of the modified program should not reveal its secrets to untrusted users. Both requirements depend on the notion of analysis, which depends on the relative precision of the observer. In this context, software diversity [6] plays a fundamental role in conjunction with code obfuscation, providing different immunity to hide vulnerabilities and enforcing that each instance must be attacked separately, dramatically increasing the effort required for hackers to develop automated attack tools [7]. Successful obfuscation should therefore implement meaningful diversity, and be able to generate diversified obfuscated code quickly enough for comparative evaluation.

1.2 The idea

We apply interpreter specialization to automatically generating obfuscated programs. An attacker (human or automatic) can only observe program execution at some fixed level of detail, e.g., in a program profiler, in a debugger, by static analysis, by program slicing, by program monitoring, or by code decompilation. In all these cases, the attacker extracts approximate properties of code under attack. We formalize the attack model as an abstract interpretation of the concrete semantics of the program. Our idea is then to transform original program P into an obfuscated program P′ by *specializing a distorted (but still correct) interpreter* as in [23, 24]. The aim is to obfuscate P either by making its observation hard and imprecise, or by making hidden information imperceptible.

The attacker. The model of attack is an approximate interpreter, formalized as an *abstract interpretation*. This general model includes relevant attack methods and tools such as: static and dynamic analysis [11], debugging [3], slicing [28], and approaches to information disclosure and reverse engineering by monitoring [13]. Obfuscation means here making the attacker unable to discern properties of code. In terms of abstract interpretation, this means that an abstract interpretion is incomplete. Completeness [12, 22] expresses precisely the accuracy of the approximate semantics in modeling program behavior. This corresponds to the possibility of replacing, with no loss of precision, concrete computations with

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

approximate ones. The lack of completeness of the attacker corresponds here to its poor understanding of the obscured program's semantics – the key for understanding the meaning of its code.

The defender. We make code protection by program transformation. Let P be the simple statement, $\mathtt{C} : \mathtt{x} = \mathtt{a} * \mathtt{b}$, multiplying a and b, and storing the result in x, as considered in [18]. An automated program *sign* analysis approximating values in the simple domain of signs is clearly able to catch, with no loss of precision, the intended behaviour of C with respect to sign, as the sign abstraction $\mathcal{O} = \{+, 0, -\}$, is complete for integer multiplication. However, if we obtain obfuscated program $\mathtt{P}'$ by replacing C with $\mathfrak{O}(\mathtt{C})$:

$\mathtt{x} = 0;\ \mathtt{if\ b} \leq 0\ \mathtt{then}\ \{\mathtt{a} = -\mathtt{a}; \mathtt{b} = -\mathtt{b}\};$
$\mathtt{while\ b} \neq 0\ \{\mathtt{x} = \mathtt{a} + \mathtt{x}; \mathtt{b} = \mathtt{b} - 1\}$

the sign analysis is now unable to extract any information concerning the computed sign, because the rule of signs is incomplete for integer addition. $\mathfrak{O}(\mathtt{C})$ is therefore an obfuscation of P for the attack performed by sign analysis. $\mathfrak{O}(\mathtt{C})$ can be obtained by modifying a self-interpreter interp in order to force abstract interpretation to deal with operations that induce incompleteness in the attacker. Suppose the distorted interpreter $\mathtt{interp}^+$ recursively implements multiplication by a sequence of additions in the obvious way (as above). This yields, by specialization as in [24], a modified program $\mathtt{P}' := [\![\mathtt{spec}]\!](\mathtt{interp}^+, \mathtt{P})$ which is precisely $\mathfrak{O}(\mathtt{C})$.

1.3 Our contribution

Three conflicts make program obfuscation a subtle problem.

1. The first concerns a general principle in programming: *good programs are well-structured and have concise invariants*. This is a key to understanding a program, and adapting it to new purposes. Good structure and short invariants are necessity in order to develop, debug and perfect a program P in the first place. However, instead *an obfuscated program should* **not** *be well-structured and easy to understand.*
2. A conflict more specific to our approach concerns performance of the obfuscated program: in [24], Section 6.4, a specializer is called "optimal" with respect to interp if, for all programs P and data d, one has for $\mathtt{P}' = [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})$: $time_{\mathtt{P}'}(\mathtt{d}) \leq time_{\mathtt{P}}(\mathtt{d})$. This asserts that specialization has *removed all interpretational overhead* from an interpreted execution $[\![\mathtt{interp}]\!](\mathtt{P}, \mathtt{d})$. Typically, optimal specialization yields a $\mathtt{P}'$ that is equivalent to P up to variable renaming, which is *not acceptable* as an obfuscation.
3. In contrast to cryptography, the current state of the art does not provide provably secure obfuscation schemata. This makes essential in any successful code protection strategy the ability to diversify code in conjunction with obfuscation, generating a large amount, at a sufficiently high rate, of different obfuscated versions of the source program. Automated code generation tools typically generate code that can be easily reverse engineered, making the diversification useless and the protection strategy vulnerable.

This paper's main contribution is in providing a comprehensive theory of systematically obscuring program analysis. The main idea is to *re-program* an interpreter interp so its specialization will obfuscate a program relatively to a given attacker. This can be achieved, given an attack specified by an abstract interpretation, by transforming the language structures and operations into new ones that defeat the considered analysis by making it incomplete. This approach to obfuscation resolves both conflicts 1 and 2 above by transforming an arbitrary program P into an equivalent $\mathtt{P}'$ in which program structure and invariants are hard to see (in code or in data). In particular, about efficiency, program $\mathtt{P}'$ may be less than "optimal", but it will still be acceptable if $time_{\mathtt{P}'}(\mathtt{d}) \leq c \cdot time_{\mathtt{P}}(\mathtt{d})$ for a reasonably small constant factor c that may depend on P, but which is independent of input data d. Moreover, by relating obscurity with interpretation, it provides an effective solution for 3, shifting the problem of generating diversified code to the problem of diversifying the interpreter. Standard results in code generation by interpreter specialization are a help in this direction.

All this is achieved by formally specifying both attack and defense strategies as complementary aspects of interpretation: the attack is an abstract interpreter designed for extracting properties of code behavior, while the defense is a program transformation that forces the attacker to follow a distorted interpretation of the transformed code, the one that increases uncertainty by maximizing the loss of precision in the abstract interpretation, thereby making the resulting program obscure.

2. Background

2.1 Basic notions

If P is a set, $\wp(P)$ is the powerset of P. $f \circ g \stackrel{\text{def}}{=} fg \stackrel{\text{def}}{=} \lambda z.\ f(g(z))$. $id \stackrel{\text{def}}{=} \lambda x.\ x$. A poset P with ordering relation $\leq$ is denoted $\langle P, \leq \rangle$, while $\langle P, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice P, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, top and bottom element $\top$ and $\bot$ respectively. Often, $\leq_P$ will be used to denote the underlying ordering of a poset P, and $\vee_P$, $\wedge_P$, $\top_P$ and $\bot_P$ denote the basic operations and elements if P is a complete lattice. Functions on ordered sets P_1 and P_2 are point-wise ordered by $\sqsubseteq$, and function $f : P_1 \longrightarrow P_2$ is additive (continuous) if f preserves *lub*'s of all subsets (chains) of P_1, empty-set included. Given a function on sets $f : P_1 \longrightarrow P_2$, its additive lift is a function $f : \wp(P_1) \longrightarrow \wp(P_2)$ such that $f(X) = \{\ f(x) \mid\ x \in X\ \}$. Additive and co-additive functions f admit respectively right and left adjoints: $f^+ \stackrel{\text{def}}{=} \lambda x.\ \bigvee \{\ y \mid\ f(y) \leq x\ \}$ and $f^- \stackrel{\text{def}}{=} \lambda x.\ \bigwedge \{\ y \mid\ x \leq f(y)\ \}$. For continuous and additive functions on a complete lattice P, the least fix-point (closure) greater or equal to $x \in P$ is denoted $lfp_x f$.

2.2 Small-step and whole-program semantics

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

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

A notational convenience: write **case** e **of** $v_1 : \mathtt{C}_1; \ldots; v_n : \mathtt{C}_n$ to stand for a chain of **if** − **then** − **else** on mutually exclusive values (the v_i are the possible values that e can take, and $\mathtt{C}_i$ is the corresponding program fragment to execute). In Table 1 we omit the formal semantics of **input** x which corresponds to a dynamic assignment to x, and of **output** x which corresponds to the *visualization* of the output result. We consider a quite standard operational semantics of the language. Let $\mathbb{P}_{\mathcal{L}}$ be a set of programs in the language $\mathcal{L}$, *Var*(P) the set of all the variables in P, and $\mathbb{PL}_{\mathtt{P}}$ be a set of program lines of $\mathtt{P} \in \mathbb{P}_{\mathcal{L}}$ containing a special notation ϵ for the empty program line, $\mathbb{V}$ be the set of values, and $\mathbb{M} \stackrel{\text{def}}{=} \mathit{Var}(\mathtt{P}) \longrightarrow \mathbb{V}$ be a set of possible program memories. When a statement st belongs to a program P we write $st \in \mathtt{P}$, then we define the auxiliary functions $\mathtt{Stm}_{\mathtt{P}} : \mathbb{PL}_{\mathtt{P}} \to \mathbb{P}_{\mathcal{L}}$ be such that $\mathtt{Stm}_{\mathtt{P}}(l) = c$ if c is the statement in P at program line l (denoted ${}^{l}\cdot c$) and $\mathtt{PC}_{\mathtt{P}} = \mathtt{Stm}_{\mathtt{P}}{}^{-1} : \mathbb{P}_{\mathcal{L}} \to \mathbb{PL}_{\mathtt{P}}$ with the simple extension to blocks of instructions $\mathtt{PC}_{\mathtt{P}}(st; \mathtt{C}) = \mathtt{PC}_{\mathtt{P}}(st)$ where $st \in \mathtt{P}$. Then, let $\sigma \in \mathbb{M}$, we define the semantics of $\mathcal{L}$ in Table 1, where x are variables, , e are expressions, B are boolean expressions, $[\![\cdot]\!]$ is the

$$\langle \sigma, \textbf{skip} \rangle \Downarrow \langle \sigma, \textbf{skip} \rangle \text{ (Fix-point)} \qquad \frac{[\![e]\!](\sigma) = n \in \mathbb{V}}{\langle \sigma, x := e \rangle \Downarrow \langle \sigma[x \mapsto n], \textbf{skip} \rangle} \qquad \frac{\langle \sigma, st \rangle \Downarrow \sigma'}{\langle \sigma, st; \mathtt{C}\ 1 \rangle \Downarrow \langle \sigma', \mathtt{C}\ 1 \rangle}$$

$$\frac{[\![B]\!]\sigma = \mathit{true}}{\langle \sigma, \textbf{if } B \textbf{ then } \mathtt{C}\ 0 \textbf{ else } \mathtt{C}\ 1 \textbf{ fi} \rangle \Downarrow \langle \sigma, \mathtt{C}\ 0 \rangle} \qquad \frac{[\![B]\!]\sigma = \mathit{false}}{\langle \sigma, \textbf{if } B \textbf{ then } \mathtt{C}\ 0 \textbf{ else } \mathtt{C}\ 1 \textbf{ fi} \rangle \Downarrow \langle \sigma, \mathtt{C}\ 1 \rangle}$$

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

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

evaluation of expressions, and where we write $\langle \sigma, \mathtt{C} \rangle \Downarrow \langle \sigma', \mathtt{C}' \rangle$ for the execution of $\mathtt{C}$ in the memory σ. We can formally characterize the small-step operational semantics of programs. Let $\mathbb{D} = \mathbb{M} \times \mathbb{P}_{\mathcal{L}}$ be the set of states, containing the actual memory and the code to execute, and $\langle \sigma, \mathtt{C} \rangle \in \mathbb{D}$. $f_{\mathcal{L}} : \mathbb{D} \longrightarrow \wp(\mathbb{D})$ is such that:

$$f_{\mathcal{L}}(\langle \sigma, \mathtt{C} \rangle) = \left\{ \langle \sigma', \mathtt{C}' \rangle \;\middle|\; \langle \sigma, \mathtt{C} \rangle \Downarrow \langle \sigma', \mathtt{C}' \rangle \right\}$$

It is worth noting that for deterministic programs, like $\mathcal{L}$, this set contains only one state. We abuse notation by denoting as $f_{\mathcal{L}}$ also its trivial additive lift on $\wp(\mathbb{D})$. We define the small-step program semantics as the fix-point of the transfer function $f_{\mathcal{L}}$: let $S \in \wp(\mathbb{D})$ then $[\![\mathtt{P}]\!](S) \stackrel{\text{def}}{=} \mathit{lfp}_S f_{\mathcal{L}} \in \wp(\mathbb{D})$. In the following, when we consider the semantics of a program P starting from any possible initial memory state of P we simply write $[\![\mathtt{P}]\!]$ denoting the set $\left\{ \mathit{lfp}_{\langle \sigma, \mathtt{P} \rangle} f_{\mathcal{L}} \;\middle|\; \sigma \in \mathbb{M} \right\}$.

Whole-program semantics: We use notations as in [24] to describe the net effect of program execution, partial evaluation, program running times, etc. Let $\mathbb{D}$ be a data set of first-order value, including *any program text* and any pair of values. The *whole-program semantic function* of program P is a partial function $[\![\mathtt{P}]\!] : \mathbb{D} \longrightarrow \mathbb{D} \cup \{\bot\}$. It transforms a program's input values into its final output (if any). If P terminates then $[\![\mathtt{P}]\!](\mathtt{d})$ is in $\mathbb{D}$, and $[\![\mathtt{P}]\!](\mathtt{d}) = \bot$ if P fails to terminate on d. Function $[\![\mathtt{P}]\!] : \mathbb{D} \longrightarrow \mathbb{D} \cup \{\bot\}$ can be obtained from the small-step semantics by fix-point closure.

We also assume given a measure of program run time (this might be proportional to the size of a computation tree deducing $[\![\mathtt{P}]\!](\mathtt{d}) = \mathtt{e}$). The time taken to compute $[\![\mathtt{P}]\!](\mathtt{d})$ is written $\mathit{time}_{\mathtt{P}}(\mathtt{d}) \in \mathbb{N} \cup \{\infty\}$; it equals ∞ iff P fails to terminate on d.

2.3 Self-interpreters and program specialization

In the following, inputs may be combined in pairs, so $(\mathtt{d}, \mathtt{e})$ is in $\mathbb{D}$ if both d and e are in $\mathbb{D}$. Program interp is a *self-interpreter* if for all programs P and data $\mathtt{d} \in \mathbb{D}$ we have $[\![\mathtt{P}]\!](\mathtt{d}) = [\![\mathtt{interp}]\!](\mathtt{P}, \mathtt{d})$. By $=$ we mean equality of partial values over $\mathbb{D} \cup \{\bot\}$.

A *partial evaluator* (or *program specializer*) is a program spec such that for every program P with "static" input $\mathtt{s} \in \mathbb{D}$ and "dynamic" input $\mathtt{d} \in \mathbb{D}$, $[\![\mathtt{P}]\!](\mathtt{s}, \mathtt{d}) = [\![[\![\mathtt{spec}]\!](\mathtt{P}, \mathtt{s})]\!](\mathtt{d})$. A specializer executes P in two stages: first, P is specialized to its first input s, yielding a "residual program" $\mathtt{P}_\mathtt{s} = [\![\mathtt{spec}]\!](\mathtt{P}, \mathtt{s})$. Second, program $\mathtt{P}_\mathtt{s}$ is run on P's dynamic input d.

A trivial specializer spec is easy to build by "freezing" the static input s (Kleene's *s-m-n* Theorem of the 1930s did specialization in this way.) The practical goal, however, is that spec should make $\mathtt{P}_\mathtt{s}$ *as efficient as possible*, by performing many of P's computations — ideally, all of those that depend on s alone. A number of practical program specializers exist. Published partial evaluation systems include *Tempo*, *Ecce*, *Logen*, *Unmix*, *Similix* and *PGG* [10, 25, 26, 32, 33]. We used Unmix in our experiments.

2.4 Program transformation by interpreter specialization

Suppose $\mathtt{P}' := [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})$ is the result of specializing a self-interpreter to program P. It is immediate that $[\![\mathtt{P}]\!] = [\![\mathtt{P}']\!]$, by simple equational reasoning: for any $\mathtt{d} \in \mathbb{D}$, we have

$$\begin{array}{lcll} [\![\mathtt{P}]\!](\mathtt{d}) & = & [\![\mathtt{interp}]\!](\mathtt{P}, \mathtt{d}) & \text{def'n of self-interpreter} \\ & = & [\![[\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})]\!](\mathtt{d}) & \text{definition of specializer} \\ & = & [\![\mathtt{P}']\!](\mathtt{d}) & \text{definition of } \mathtt{P}' \end{array}$$

Therefore function $\lambda \mathtt{P}.[\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})$ is a semantics-preserving program transformer [23]. However, even though P and P′ are *semantically* equivalent, they may be *syntactically* quite different (far more different than just by renaming). Examples in [23] show why

1. program P′ inherits the *programming style* of interp; but
2. program P′ inherits the *algorithm* of program P;.

The reason for Point 1 is that program P′ is specialized code taken from the interpreter interp: P′ consists of the operations of interp that depend on its dynamic input d (and not only on P, else they would have been "specialized away"). A *general-purpose program transformer* can thus be built by programming self-interpreter interp in a style appropriate to the desired transformation.

The reason for Point 2 is that even though program P′ may be a disguised form of P, a correct interpreter interp must faithfully execute the operations that P specifies. In usual practice, the specialized program P′ will perform *the same computations in the same order* as those performed by P. One does not expect an interpreter to devise new computational approaches.

3. A top-down view of program obfuscation

The point of program obfuscation is to transform a program P, or parts of a program, into a new program P′ that is computationally equivalent, but such that P′ is hard to adapt or analyze. Criteria for good obfuscation are:

1. *(Semantics) Preservation*: $[\![\mathtt{P}']\!] = [\![\mathtt{P}]\!]$ is necessary.
2. *Automation*: P′ is obtained from P without hand work.
3. *Potency*: P is hard to obtain from P′. The goal is that, even though P′ can be executed, it should be hard to adapt, exploit, or analyze by automatic and manual methods.
4. *Efficiency*: program P′ should not be too much slower or larger than P. It is acceptable, however, for interp to be slow, as long as P′ is satisfactorily efficient and hard to understand.

To satisfy these criteria, we design interp so that in general P′ will be harder to abstractly interpret than P is; and P′ will be satisfactorily efficient. We prove that many useful program obfuscations can be obtained by interpreter specialization, achieving (1) and (2) straight from correctness of the self-interpreter and the specializer.

We exemplify this in the case of *code flattening*. The idea is simple: in conventional program analysis, without knowledge of the branch targets and the execution order of the code blocks, every block is potentially the immediate predecessor of every other block, making the overall control-flow obscure. Code flattening is the basic principle behind relevant obfuscation strategies, most devoted to make hard the reverse engineering of flattening, such as obfuscation by unintelligible (NP-hard) dispatcher design in [4] and obfuscation of the dispatcher by enlarging its search space with spurious aliases in [34]. We begin with simple self-interpreter `interp`, and then modify `interp` into an equivalent $\texttt{interp}^{\texttt{flat}}$ whose specialization will, for any P, yield $\texttt{P}' := [\![\texttt{spec}]\!](\texttt{interp}^{\texttt{flat}}, \texttt{P})$ with non-evident (hidden) control flow. The general idea is to *reprogram* `interp` so its specialization will "scramble" or "distort" the control-flow and/or data-flow of its input program P, without changing its whole-program semantics.

EXAMPLE 3.1. **Flattening:** *consider the program* P *on the left. A flattened equivalent program (on the right) has an explicit program counter* pc *and only one loop, regardless of* P.

```
1· input x;                    1· input x; 2· pc := 2;
2· y := 2;                     3· while pc < 6 do
3· while x > 0 do                 4· case pc of
       4· y := y + 2;                 2 : 5· y := 2; 6· pc := 3;
       5· x := x − 1                  3 : 7· if x > 0 then 8· pc := 4 else 9· pc := 6 fi;
   endw                               4 : 10· y := y + 2; 11· pc := 5;
6· output y;                          5 : 12· x := x − 1; 13· pc := 3;
7· end                             endw
                                   14· output y
                                   15· end
```

Experimental context: We did the following and other experiments using the Unmix specializer (suitable since its input language SCHEME is good for program representations). Both `interp` and `spec` are general SCHEME programs. However, their input and output P and P′ are tail-recursive programs, in a SCHEME subset isomorphic to the imperative language $\mathcal{L}$ from Section 2.2.

Structure of the simple self-interpreter. `interp` is a SCHEME implementation of the following. Assume input program P has variables $in, out, x_1, \ldots, x_n$. The overall structure of the interpreter is traditional, with a "dispatch on syntax" loop: find the form of the current P instruction at pc (program counter), and then execute it. The memory of program P is held in `interp` variable $store$.

```
input P, d;                                  Program to be interpreted, and its data
pc := 2; store := [in ↦ d, out ↦ 0, x1 ↦ 0, ...];
while pc < length(P) do
    instruction := lookup(P, pc);            Find the pc-th instruction
    case instruction of                      Dispatch on syntax
    skip     : pc := pc + 1;
    x := e   : store := store[x ↦ eval(e, store)]; pc := pc + 1;
    ...  endw ;
output store[out];
eval(e, store) = case e of                   Function to evaluate expressions
    constant : e
    variable : store(e)
    e1 + e2  : eval(e1, store) + eval(e2, store)
    e1 − e2  : eval(e1, store) − eval(e2, store)
    e1 ∗ e2  : eval(e1, store) ∗ eval(e2, store)
    ...
end
```

Specialization of the simple self-interpreter: This gives a residual (specialized) program $\texttt{P}' = [\![\texttt{spec}]\!](\texttt{interp}, \texttt{P})$ that is essentially identical to P (up to variable renaming), as follows:

- `interp` variable P is classified as "static", and variable d is classified as "dynamic".
- `interp` variables e, pc and $instruction$ are classified as "static", since given any P they can only assume finitely many values.
- `interp` function $eval$ is completely unfolded and so does not appear in P′, since all recursive calls decrease the static value e.
- The simple self-interpreter's **while** loop is completely unfolded by Unmix, so the only remaining control transfers correspond to those present in program P.
- `interp` variable $store$ is a function with static domain but dynamic range. Unmix does "arity raising", splitting $store$ into one residual program variable for each of P's variables.

Specialization of the "flattening" interpreter: A small change to `interp` gives the residual program $\texttt{P}' = [\![\texttt{spec}]\!](\texttt{interp}^{\texttt{flat}}, \texttt{P})$ as in Example 3.1. The essential trick here is to recode `interp` so that the specializer will classify variable pc as *dynamic*.[1] Since pc is dynamic, the **while** loop in $\texttt{interp}^{\texttt{flat}}$ will no longer be unfolded by Unmix, and so comes to appear in the specialized program P′. This accounts for the form of P′ in Example 3.1. The transformation $\texttt{P} \mapsto \texttt{P}' = [\![\texttt{spec}]\!](\texttt{interp}^{\texttt{flat}}, \texttt{P})$ will flatten *any* $\mathcal{L}$ program; i.e., it is in no way specific to this example input program P.

4. The potency of obfuscation

The design of potent obfuscations requires a precise model of attacker. We consider the potency of a program-transforming obfuscation to be its ability to make imprecise an approximate semantics, viz. an abstract interpreter, following the idea introduced in [18].

4.1 Abstract interpretation

For simplicity we consider Galois connection based abstract interpretations [11, 12]. An abstraction on a domain D, partially ordered by $\leq_D$, is any function $\rho : D \longrightarrow D$ such that $x \leq_D \rho(x)$, meaning that $\rho(x)$ contains less information than x, ρ is monotone, meaning that it respects the relative precision of objects, and $\rho(\rho(x)) = \rho(x)$ meaning that the loss of information made by abstraction is performed *all-at-once*. Abstractions and abstract domains can in this way be isomorphically represented as *upper closure operators* on a concrete domain D, i.e., elements in $uco(D)$ [12]. Closure operators in $uco(D)$ are uniquely determined by the set of their fix-points $\rho(D)$. $X \subseteq C$ is the set of fix-points of $\rho \in uco(D)$ iff X is a *Moore-family* of D, i.e., $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$, $\wedge\varnothing = \top \in \mathcal{M}(X)$. If D is a complete lattice then $\langle uco(D), \sqsubseteq\rangle$ is also a complete lattice $\langle uco(D), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, id\rangle$, where for any $\rho, \eta \in uco(D)$, $\{\rho_i\}_{i\in I} \subseteq uco(D)$ and $x \in D$: $\rho \sqsubseteq \eta$ iff $\eta(D) \subseteq \rho(D)$; $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$; and $(\sqcup_{i\in I}\rho_i)(x) = x \Leftrightarrow \forall i \in I.\ \rho_i(x) = x$. In the following we will find convenient to identify closure operators ρ equivalently either as functions (abstractions) or as sets $\rho(D)$ (abstract domains).

Soundness. Soundness means that abstract computation includes all possible concrete computations: the semantic approximation is from above [12]. Let $f : D \xrightarrow{c} D$ be a continuous predicate transformer (also called *transfer function*), $\rho, \eta \in uco(D)$, and $f^\sharp : \eta(D) \xrightarrow{m} \rho(D)$. Then $f^\sharp$ is a sound abstraction of f from η to ρ if $\rho \circ f \leq f^\sharp \circ \eta$. The *best correct approximation* of f is $f^{bca} \stackrel{\text{def}}{=} \rho \circ f \circ \eta$. It is known [12] that $f^\sharp$ is sound iff $f^{bca} \sqsubseteq f^\sharp$; implying $\rho(lfp(f)) \leq lfp(f^{bca}) \leq lfp(f^\sharp)$. Suppose $[\![\texttt{P}]\!]$ is specified as fix-point of (a combination of) predicate-transformers $F_\texttt{P} : D \xrightarrow{c} D$, and $\rho, \eta \in uco(D)$. Define $[\![\texttt{P}]\!]^{(\rho,\eta)}$ to be the abstract

[1] Technically, pc is made dynamic using the Unmix `generalize` annotation. Some modest reprogramming is also needed: `interp` is extended to have both dynamic and static copies of pc, so specialization will generate P′ code such as **case** $\ldots$ $pc = 5 :\ x := x - 1; pc := 3$ in Example 3.1.

semantics associated with $F^{bca}_{\mathtt{P}}$, so $[\![\mathtt{P}]\!]^{(\rho,\eta)}$ is the *best correct abstract interpretation of* $\mathtt{P}$ *in* ρ *and* η. In this case soundness means $\rho([\![\mathtt{P}]\!]) \leq [\![\mathtt{P}]\!]^{(\rho,\eta)}$.

Completeness. Completeness means that no loss of precision is added by computing with approximate objects [12, 22], namely $\rho([\![\mathtt{P}]\!]) = [\![\mathtt{P}]\!]^{(\rho,\eta)}$. Following [21] we distinguish between *backward* ($\mathscr{B}$) and *forward* ($\mathscr{F}$) completeness. Backward completeness holds when $\rho \circ f = f^\sharp \circ \eta$, meaning that no loss of precision is accumulated by approximating the input arguments of a concrete transfer function. Forward completeness holds when $f \circ \eta = \rho \circ f^\sharp$, meaning that no loss of precision is accumulated by approximating the result of computations on abstract objects. The key point in this construction is that there exists an either $\mathscr{B}$ or $\mathscr{F}$-complete abstract function $f^\sharp$ for $\rho, \eta \in \mathit{uco}(C)$ iff the best correct approximation $\rho \circ f \circ \eta$ of f is respectively either $\mathscr{B}$ or $\mathscr{F}$-complete, i.e., $\rho \circ f = \rho \circ f \circ \eta$ or $f \circ \eta = \rho \circ f \circ \eta$ [22]. This means that both $\mathscr{F}$ and $\mathscr{B}$ completeness are properties of the abstraction in relation with concrete predicate transformers. It is known [22] that, if f is additive: $\rho \circ f = \rho \circ f \circ \eta$ if and only if $f^+ \circ \rho = \eta \circ f^+ \circ \rho$.

Program properties. It is known that any abstraction on a concrete domain D, i.e., a pair of closures $\rho, \eta \in \mathit{uco}(D)$, induces a corresponding partition on programs in $\mathbb{P}_{\mathcal{L}}$:

$$\mathtt{P} \equiv_{\rho,\eta} \mathtt{Q} \iff [\![\mathtt{P}]\!]^{(\rho,\eta)} = [\![\mathtt{Q}]\!]^{(\rho,\eta)}$$

It is clear (e.g., see [12]) that the equivalence induced by the concrete semantics is a refinement of any partition induced by a sound abstract interpretation and that, by refining abstractions, e.g., by completeness [22], we obtain refined partitions of programs. In the following we identify the closures ρ and η with the property on programs corresponding to the above partition, and call ρ and η respectively output and input program properties.

4.2 Example: attack model foiled by Flattening

In this section, we show that the control flow flattening-based obfuscation can be modeled as a problem of making incomplete an abstract interpreter. The attacker is an abstract interpreter extracting the control flow graph, which is obtained by considering two abstraction steps: in input we lose the control flow when the program counter is dynamic, namely when it is controlled in the program itself, in output we lose the memory and the history of computations (traversed branches).

Graph semantics. Consider the set of graphs defined as follows: $\mathtt{G} = \langle \mathit{Nodes}(\mathtt{G}), \mathit{Arcs}(\mathtt{G}) \rangle \in \mathbb{G}$, where $\mathit{Nodes}(\mathtt{G}) \subseteq \mathbb{PL}_{\mathtt{P}}$ and $\mathit{Arcs}(\mathtt{G}) \subseteq \mathbb{PL}_{\mathtt{P}} \times \mathbb{PL}_{\mathtt{P}}$. In order to characterize the history of computation, we define the function determining the sequence of program lines executed: $\mathtt{Next}_{\mathtt{P}} : \mathbb{M} \times \mathbb{PL}_{\mathtt{P}} \to \mathbb{PL}_{\mathtt{P}}$.

$$\begin{array}{l} \mathtt{Next}_{\mathtt{P}}(\sigma, l) = l' \qquad \text{iff} \\ \quad f_{\mathcal{L}}(\langle \sigma, \mathtt{Stm}_{\mathtt{P}}(l) \rangle) = \langle \sigma', \mathtt{C} \rangle \ \wedge \ \mathtt{Pc}_{\mathtt{P}}(\mathtt{C}) = l' \end{array}$$

Let us define the auxiliary function: $\mathtt{Next}^\sharp_{\mathtt{P}} : \mathbb{PL}_{\mathtt{P}} \to \wp(\mathbb{PL}_{\mathtt{P}})$:

$$\mathtt{Next}^\sharp_{\mathtt{P}}(l) = \begin{cases} \{\mathtt{Pc}_{\mathtt{P}}(\mathtt{C}), l + |\mathtt{C}| + 1\} \\ \qquad \text{if } \mathtt{Stm}_{\mathtt{P}}(l) = \textbf{while } B \textbf{ do } \mathtt{C} \textbf{ endw} \\ \{\mathtt{Pc}_{\mathtt{P}}(\mathtt{C}_1), \mathtt{Pc}_{\mathtt{P}}(\mathtt{C}_2)\} \\ \qquad \text{if } \mathtt{Stm}_{\mathtt{P}}(l) = \textbf{if } B \textbf{ then } \mathtt{C}_1 \textbf{ else } \mathtt{C}_2 \textbf{ fi} \\ \{\, \mathtt{Next}_{\mathtt{P}}(l, \sigma) \mid \sigma \in \mathbb{M} \,\} \\ \qquad \text{otherwise} \end{cases}$$

Note that, $\mathtt{Next}^\sharp_{\mathtt{P}}$ can be defined as abstract interpretation of $\mathtt{Next}_{\mathtt{P}}$, where we abstract on the evaluation of the guard of **while** and **if**. Let us define the graph semantics of a program $\mathtt{P}$:

- **States**: $\mathbb{D}_{\mathbb{G}} = \mathbb{M} \times ((\mathbb{N} \cup \{\epsilon\}) \times \mathbb{N}) \times \mathbb{G}$. A state is $\langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle$ where $\sigma \in \mathbb{M}$ is the memory before the execution of l_2, l_1 is the program line of the last executed statement, and $\mathtt{G}$ represents the execution graph until the execution of l_1 (l_1 included).
- **Transfer functions:** $g_{\mathcal{L}} : \mathbb{D}_{\mathbb{G}} \longrightarrow \mathbb{D}_{\mathbb{G}}$: $g_{\mathcal{L}} \stackrel{\text{def}}{=} \lambda\langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle.\ \langle \sigma', \langle l_2, \mathtt{Next}_{\mathtt{P}}(\sigma, l_2) \rangle, \mathtt{G} \uplus \{l_1, l_2\} \rangle$: where $\sigma' = f_{\mathcal{L}}(\langle \sigma, \mathtt{Stm}_{\mathtt{P}}(l_2) \rangle)_{|\mathbb{M}}$ and $\langle \sigma, \mathtt{C} \rangle_{|\mathbb{M}} = \sigma$, $\uplus$ denotes the union of graphs, i.e., $\mathtt{G}_1 \uplus \mathtt{G}_2 = \mathtt{G}$ where we have $\mathit{Nodes}(\mathtt{G}) \stackrel{\text{def}}{=} \mathit{Nodes}(\mathtt{G}_1) \cup \mathit{Nodes}(\mathtt{G}_2)$ and $\mathit{Arcs}(\mathtt{G}) \stackrel{\text{def}}{=} \mathit{Arcs}(\mathtt{G}_1) \cup \mathit{Arcs}(\mathtt{G}_2)$, and $\{l_1, l_2\}$ is a simplified notation for the graph $\mathtt{G}'$ such that $\mathit{Nodes}(\mathtt{G}') = \{l_1, l_2\}$ and $\mathit{Arcs}(\mathtt{G}') = \{\langle l_1, l_2 \rangle\}$. We abuse notation denoting by $g_{\mathcal{L}}$ also its additive lift on $\wp(\mathbb{D}_{\mathbb{G}})$.

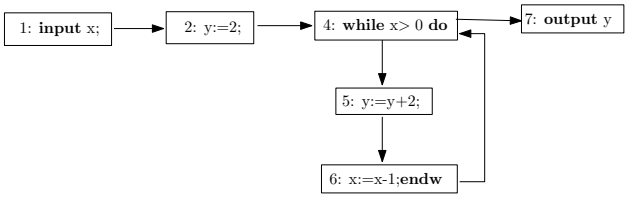
In graph semantics $\langle l_1, l_2 \rangle \in \mathit{Arcs}(\mathtt{G})$ iff $l_2 \in \mathtt{Next}^\sharp_{\mathtt{P}}(l_1)$. For any program $\mathtt{P} \in \mathbb{P}_{\mathcal{L}}$, its generic initial state is $s_{\mathtt{P}} = \langle \sigma, \langle \epsilon, 1 \rangle, \langle \varnothing, \varnothing \rangle \rangle \in \mathbb{D}_{\mathbb{G}}$, where $\langle \epsilon, 1 \rangle$ is the initial point of the program corresponding to program line 1. We can define the graph semantics in fix-point form: $[\![\mathtt{P}]\!]_{\mathbb{G}}(s_{\mathtt{P}}) \stackrel{\text{def}}{=} \mathit{lfp}_{s_{\mathtt{P}}}\, g_{\mathcal{L}} \in \wp(\mathbb{D}_{\mathbb{G}})$. Also in this case, when we consider the semantics of a program $\mathtt{P}$ starting from any possible initial state of $\mathtt{P}$ we simply write $[\![\mathtt{P}]\!]_{\mathbb{G}}$ to denote the set $\{\mathit{lfp}_{s_{\mathtt{P}}}\, g_{\mathcal{L}} \mid \sigma \in \mathbb{M}\}$. In the following we denote by $\sqsubseteq_{\mathbb{G}}$ the standard subgraph relation, i.e., $\mathtt{G}_1 \sqsubseteq_{\mathbb{G}} \mathtt{G}_2$ iff $\mathit{Nodes}(\mathtt{G}_1) \subseteq \mathit{Nodes}(\mathtt{G}_2)$ and $\mathit{Arcs}(\mathtt{G}_1) \subseteq \mathit{Arcs}(\mathtt{G}_2)$. For simplicity we abuse notation by using the same relation between states, i.e., $\langle \sigma_1, \langle l'_1, l''_1 \rangle, \mathtt{G}_1 \rangle \sqsubseteq_{\mathbb{G}} \langle \sigma_2, \langle l'_2, l''_2 \rangle, \mathtt{G}_2 \rangle$ iff $\mathtt{G}_1 \sqsubseteq_{\mathbb{G}} \mathtt{G}_2$. The next result proves that $g_{\mathcal{L}}$ is increasing.

PROPOSITION 4.1. *Let* $s \in \mathbb{D}_{\mathbb{G}}$. $\forall n \in \mathbb{N}.\ g^n_{\mathcal{L}}(s) \sqsubseteq_{\mathbb{G}} g^{n+1}_{\mathcal{L}}(s)$.

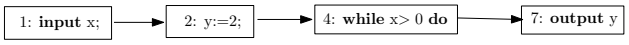
EXAMPLE 4.2. *Consider the program in Example* 3.1. *In this case the small-step semantics when* $x = 1$ *is:*

$$\begin{array}{l} \langle (x, y), \langle \epsilon, 1 \rangle, \langle \varnothing, \varnothing \rangle \rangle \longrightarrow \langle (1, y), \langle 1, 2 \rangle, \mathtt{G}_1 \rangle \\ \qquad \longrightarrow \langle (1, 2), \langle 2, 3 \rangle, \mathtt{G}_2 \rangle \longrightarrow \langle (1, 2), \langle 3, 4 \rangle, \mathtt{G}_3 \rangle \\ \qquad \longrightarrow \langle (1, 4), \langle 4, 5 \rangle, \mathtt{G}_4 \rangle \longrightarrow \langle (0, 4), \langle 5, 3 \rangle, \mathtt{G}_5 \rangle \\ \qquad \longrightarrow \langle (0, 4), \langle 3, 6 \rangle, \mathtt{G}_6 \rangle \longrightarrow \langle (0, 4), \langle 6, 7 \rangle, \mathtt{G}_7 \rangle \end{array}$$

The final graph $\mathtt{G}_7$ *is:*



Note that, if the input for x *is* 0*, then the guard of the* **while** *becomes false and therefore the semantics and the final graph is:*



Abstracting the program control structure. Flattening means to lose the control structure of a program only when it controls the program counter. In other words, an observer is unable to understand which is the next statement of each program point. We observed that this can be achieved by considering the program counter ($\mathtt{pc}$) as a program variable, which is determined dynamically instead of statically. In this way, the control flow constructed corresponds to the real execution (branch) when the control statement guard concerns only program variables, but it loses the branch information when among the controlled variables there is the program counter. The next function is a recursive map that skips the evaluation of the guards concerning the program counter. The recursive function $\mathcal{F}$ decides when to skip the evaluation of the guard, while $\mathcal{B}$ takes both the branches of a control structure ignoring the evaluation of the corresponding guard. We need a recursive application since we may have nested control structures to check. Let

$\langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle \in \mathbb{D}_{\mathbb{G}}$:

$$\mathcal{F}(\langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle) \stackrel{\text{def}}{=} \begin{cases} \mathcal{F} \circ \mathcal{B}(\langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle) & \text{if } \mathtt{Stm_P}(l_2) \in \{\textbf{if } B, \textbf{while } B\} \wedge \\ & \quad \mathtt{pc} \in \mathit{Var}(B) \\ \langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle & \text{otherwise} \end{cases}$$

$$\mathcal{B}(\langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G} \rangle) = \left\{ \langle \sigma, \langle l_2, l \rangle, \mathtt{G} \uplus \{l_1, l_2\} \rangle \;\middle|\; l \in \mathtt{Next}^{\sharp}_{\mathtt{P}}(l_2) \right\}$$

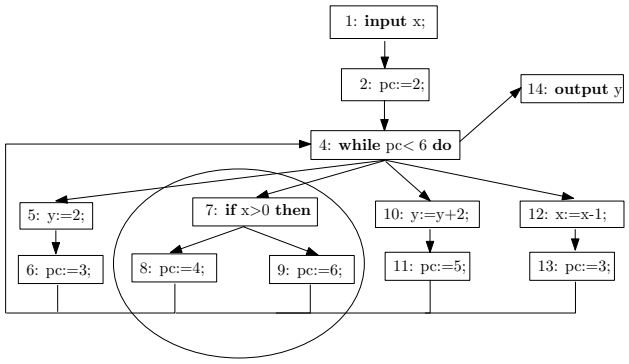
Let us denote by $\mathcal{F}$ also its fix-point closure.

PROPOSITION 4.3. $\mathcal{F} \in \mathit{uco}(\wp(\mathbb{D}_{\mathbb{G}}))$.

EXAMPLE 4.4. *Consider the flattened program of Example* 3.1. *A recursive definition of $\mathcal{F}$ is needed because we have to apply $\mathcal{B}$ to a* **case** *statement on* pc*, nested in a* **while** *also controlling* pc*:*

$$\begin{aligned} &\mathcal{F} \circ \mathcal{B}(\langle\langle (1,2), \langle 2,3 \rangle, \mathtt{G}_2 \rangle\rangle) \\ &\quad = \mathcal{F}(\{\langle\langle (1,2), \langle 3,4 \rangle, \mathtt{G}'_3 \rangle, \langle (1,2), \langle 3,14 \rangle, \mathtt{G}''_3 \rangle\rangle\}) \\ &\quad = \{\mathcal{F} \circ \mathcal{B}(\langle\langle (1,2), \langle 3,4 \rangle, \mathtt{G}'_3 \rangle\rangle), \langle (1,2), \langle 3,14 \rangle, \mathtt{G}''_3 \rangle\} \\ &\quad = \left\{ \begin{array}{l} \mathcal{F}(\{\, \langle (1,2), \langle 4,i \rangle, \mathtt{G}_i \rangle \;|\; i \in \{5,7,10,12\} \,\}), \\ \langle (1,2), \langle 3,14 \rangle, \mathtt{G}''_3 \rangle \end{array} \right\} \\ &\quad = \left\{ \begin{array}{l} \{\, \langle (1,2), \langle 4,i \rangle, \mathtt{G}_i \rangle \;|\; i \in \{5,7,10,12\} \,\}, \\ \langle (1,2), \langle 3,14 \rangle, \mathtt{G}''_3 \rangle \end{array} \right\} \end{aligned}$$

Note that, $\mathcal{F}(\langle (1,2), \langle 4,7 \rangle, \mathtt{G}_7 \rangle) = \langle (1,2), \langle 4,7 \rangle, \mathtt{G}_7 \rangle$ *because the statement at program line* 7 *is an* **if***, whose guard does not concern the program counter. The resulting control flow-graph is:*



Note that the circled part will select a branch depending on the value of x, namely it will correspond to the real computation.

Abstracting the memory. We define the output abstraction losing the memory and the branch computational history. Let $S \subseteq \mathbb{D}_{\mathbb{G}}$.

$$\mathcal{C}(S) \stackrel{\text{def}}{=} \left\{ \langle M, \langle l_1, l_2 \rangle, \bigoplus_i \mathtt{G}_i \rangle \;\middle|\; \forall \sigma \in M. \exists i.\ \langle \sigma, \langle l_1, l_2 \rangle, \mathtt{G}_i \rangle \in S \right\}$$

This function abstracts the memory and the program structure component-wise, keeping no relational information among them. Note that, this function is trivially an abstraction on $\mathbb{D}_{\mathbb{G}}$.

PROPOSITION 4.5. $\mathcal{C} \in \mathit{uco}(\wp(\mathbb{D}_{\mathbb{G}}))$.

Flattening as incompleteness. At this point the abstract semantics losing the control structure on the program counter can be specified as an abstract interpretation: $[\![\mathtt{P}]\!]_{\mathcal{F}} = \mathit{lfp}\ (g^{\mathcal{F}}_{\mathcal{L}})_{|_{\mathbb{G}}}$ where $g^{\mathcal{F}}_{\mathcal{L}} \stackrel{\text{def}}{=} \mathcal{C} \circ g_{\mathcal{L}} \circ \mathcal{F}$. The next result justifies the construction of the distorted interpreter $\mathtt{interp}^{\mathtt{flat}}$ in Section 3: *Flattening the control structure of a program corresponds precisely to dynamically interpreting the program counter, which is equivalent to making incomplete the abstract interpreter that collects the CFG and does not evaluate guards concerning the program counter.*

THEOREM 4.6. $\mathcal{C}([\![\mathtt{P}]\!]_{\mathbb{G}}) =_{\mathbb{G}} [\![\mathtt{P}]\!]_{\mathcal{F}}$ *iff* P *has no dynamic* pc.

5. Incompleteness driven obfuscation

Let $\rho, \eta \in \mathit{uco}(\wp(\mathbb{D}))$ be properties of program execution states, and let $\mathfrak{O} : \mathbb{P} \longrightarrow \mathbb{P}$ be a program transformation such that

(1): $[\![\mathtt{P}]\!] = [\![\mathfrak{O}(\mathtt{P})]\!]$. Assume that (ρ, η) is $\mathscr{B}$-complete for $[\![\mathtt{P}]\!]$, i.e.,

(2): $\rho([\![\mathtt{P}]\!]) = [\![\mathtt{P}]\!]^{(\rho,\eta)} \stackrel{\text{def}}{=} \left\{ \mathit{lfp}_{\langle \sigma, \mathtt{P} \rangle} \rho \circ f_{\mathcal{L}} \circ \eta \;\middle|\; \sigma \in \mathbb{M} \right\}$. Then $\mathfrak{O}$ obfuscates P for the property (ρ, η) if

(3): $[\![\mathtt{P}]\!]^{(\rho,\eta)} \sqsubset [\![\mathfrak{O}(\mathtt{P})]\!]^{(\rho,\eta)}$.

THEOREM 5.1 ([18]). *For any $[\![\cdot]\!] : \mathbb{P} \longrightarrow \wp(\mathbb{D})$, $\rho, \eta \in \mathit{uco}(\wp(\mathbb{D}))$, transformation $\mathfrak{O} : \mathbb{P} \longrightarrow \mathbb{P}$, and $\mathbb{P}$ program* P *we have:*

$$[\![\mathtt{P}]\!]^{(\rho,\eta)} \sqsubset [\![\mathfrak{O}(\mathtt{P})]\!]^{(\rho,\eta)} \ \textit{ iff } \ \rho([\![\mathfrak{O}(\mathtt{P})]\!]) \sqsubset [\![\mathfrak{O}(\mathtt{P})]\!]^{(\rho,\eta)}.$$

The intuition here is simple: obfuscating a program means making an observer (the attacker) unable to observe some aspects of the computation, thereby lessening its ability to distinguish programs by observing what and how they compute. The observable property (ρ, η) here is fixed and cannot be distorted, therefore we can only act by distorting the program's syntax so that a loss of precision is induced in the abstract interpretation of the distorted program. We now prove that such a distorted program can be systematically obtained by specializing a distorted interpreter, in such a way that the residual program makes an abstract interpretation incomplete.

5.1 Universal distortion: obfuscator generation

THEOREM 5.2. *Suppose* int *is any correct self-interpreter for language $\mathcal{L}$. Then $[\![\mathtt{P}]\!] = [\![\mathfrak{O}(\mathtt{P})]\!]$ holds for $\mathfrak{O}(\mathtt{P}) \stackrel{\text{def}}{=} [\![\mathtt{spec}]\!](\mathtt{int}, \mathtt{P})$.*

An obfuscator can therefore be generated by the following steps:

- Consider a $\mathscr{B}$-complete property (ρ, η) on the semantics of program P, where ρ/η are the observable output/input properties;
- Isolate a set of programs, here called *incomplete structures*

$$\Pi = \left\{ \mathtt{P} \in \mathbb{P}_{\mathcal{L}} \;\middle|\; \rho([\![\mathtt{P}]\!]) \neq [\![\mathtt{P}]\!]^{(\rho,\eta)} \right\}$$

- Design a correct self-interpreter $\widetilde{\mathtt{Int}}$ such that $\mathfrak{O}(\mathtt{P}) \in \Pi$ for some $\mathcal{L}$-programs P.

Now (1) holds by Theorem 5.2, and (2) since (ρ, η) is $\mathscr{B}$-complete. (3) will hold if $\widetilde{\mathtt{Int}}$ can be devised such that $\mathfrak{O}(\mathtt{P}) \in \Pi$ for some $\mathcal{L}$-programs P. In practice this can be straightforwardly achieved by modifying a "vanilla" self-interpreter int. We have already seen one special case of universal distortion, namely Example 3.1. For this flattening example, $\widetilde{\mathtt{Int}}$ uses a dynamic variable pc, representing a program counter. Since pc is dynamic it appears in the specialized programs $\mathfrak{O}(\mathtt{P})$; so the incomplete structures in Π are one-loop programs with control statements incorporating pc.

The following example (more directly following the recipe above) does a workaround in int so that its specialised code avoids the operator $*$ on which sign analysis is $\mathscr{B}$-complete.

EXAMPLE 5.3. *Consider the simple case of sign analysis specified by the abstraction $\rho_s \in \mathit{uco}(\wp(\mathbb{Z}))$: $\rho_s = \{\varnothing, 0, 0+, 0-, \mathbb{Z}\}$ where $0+$ and $0-$ represent respectively the set of all non negative and non positive integers. A distorted interpreter can be obtained by replacing in the interpreter for the language of call-by-value recursion equation systems the evaluation of integer multiplication by calling a function to multiply by repeated addition.*

$$\begin{array}{l} \mathit{eval}(e, \mathit{env}) \ = \ \textbf{case } e \textbf{ of} \\ \quad e1 * e2 \ : m(\mathit{eval}(e1, \mathit{env}), \mathit{eval}(e2, \mathit{env})) \\ \quad \ldots \\ \textbf{end} \\ m(a,b) \ = \ \textbf{if } b < 0 \textbf{ then } mm(-a, -b, 0) \textbf{ else } mm(a, b, 0) \textbf{ fi} \\ mm(a,b,z) \ = \ \textbf{if } b = 0 \textbf{ then } z \textbf{ else } mm(a, b-1, z+a) \textbf{ fi} \end{array}$$

Specialization of this distorted interpreter will yield only specialized programs written in the following simple functional programming language $\mathcal{L}^+$ with $\mathtt{n} \in \mathbb{Z}$:

```
Program  ::= Equation,...,Equation
Equation ::= FuncName(Varlist) = Exp
Varlist  ::= Var,...,Var
Exp ::=  n  |  Var  |  if Exp then Exp else Exp
     |   FuncName(Arglist)
     |   Exp+Exp | Exp-Exp | Exp<Exp | Exp=Exp
Arglist  ::= Exp,...,Exp
```

This grammar generates only programs without $*$. These are incomplete *with respect to the abstract interpretation $[\![\_]\!]^{(\rho_s,\rho_s)}$ obtained by abstracting with the rule of signs the language of call-by-value recursion equation systems. The interpreter operates on an environment defined as a function env mapping variables into $\mathbb{Z}$. The corresponding abstract environment maps variables into ρ_s.*

If given as static input program $\mathtt{P} : 2mult(x,y) = y*x+x*y$ and as dynamic input x and y, the Unmix specializer will return (modulo some syntactic sugar) the following obfuscated program:

$$\begin{array}{l} 2mult(x,y) \;=\; m(y,x) + m(x,y) \\ m(a,b) \;=\; \textbf{if } b < 0 \textbf{ then } mm(-a,-b,0) \textbf{ else } mm(a,b,0) \textbf{ fi} \\ mm(a,b,z) \;=\; \textbf{if } b = 0 \textbf{ then } z \textbf{ else } mm(a,b-1,z+a) \textbf{ fi} \end{array}$$

In this case, $[\![\mathtt{P}]\!]^{(\rho_s,\rho_s)}(x,y) \neq \mathbb{Z}$ iff $x = 0$ or $y = 0$, making the abstract interpreter of the transformed programs unable to extract the sign of the result of the computation.

5.2 Local distortion

In this section we explore weaker conditions for the existence of a distorted interpreter. The idea is to act locally, to restricted portions of the source program, making them obscure with respect to a fixed abstraction, yet inducing obscurity in the resulting program. This is indeed a common practice in code obfuscation, where obfuscation is restricted to specific code portions, e.g., those holding secrets relevant for the whole program structure. Let us consider a program $\mathtt{P}$, and consider a set of its variables, $\{x_i\}_{i\in[1,n]}$. If $\mathtt{P}_{[x]}$ denotes the presence of a variable x in $\mathtt{P}$, its β-reduction obtained by substituting x with a value $d \in \mathbb{V}$ in $\mathtt{P}$ is denoted $\mathtt{P}_{[d]}$. For each i, let $\mathtt{P}_i$ be the program fragment of $\mathtt{P}$ that computes the value of x_i during the execution of the program $\mathtt{P}$. Hence we use the notation $\mathtt{P}_{[\mathtt{P}_1 \ldots \mathtt{P}_n]}$ for representing these elements in $\mathtt{P}$ and we denote by $\mathtt{P}_{[x_1 \ldots x_n]}$ ($\mathtt{P}_{[x_i]}$ for short) the program $\mathtt{P}$ where all the program fragments $\mathtt{P}_i$ are substituted by **skip**. This can be achieved for instance by a slicing algorithm [31, 35] that extracts $\mathtt{P}_i$ out of $\mathtt{P}$ by slicing $\mathtt{P}$ with respect to some of its (local) variables x_i. If we consider a family $\{\mathtt{Q}_i\}_{i\in[1,n]}$ to substitute to the several $\mathtt{P}_i$ we denote the resulting program as $\mathtt{P}_{[\mathtt{Q}_1 \ldots \mathtt{Q}_n]}$ ($\mathtt{P}_{[\mathtt{Q}_i]}$ for short).

The effect of replacing code in programs can only be understood by the notion of dependency [1]. We model the effect of substituting program fragments with *equivalent* incomplete structures by *abstract non-interference* [19]. Abstract non-interference (ANI) is a natural weakening of non-interference (and therefore of dependency) by abstract interpretation, providing a model where interfering objects are properties of code instead of actual computed values. We want to characterize the *dependency* between the computation of the values of the different variables x_i and the observable semantics of the program $\mathtt{P}$. For this reason we define the following notion of *stability*, which specifies that a program $\mathtt{P}$ is stable w.r.t. the variables x_i, when any change of its abstract (computed) values does not change the observable semantics of the program. First of all, consider as observable semantics what the attacker can see, namely the abstract semantics computed by abstract interpretation $[\![\mathtt{P}]\!]^{(\rho,\rho)}$, with $\mathbb{D} = \mathbb{M} = Var \longrightarrow \mathbb{V}$ and where we abuse notation by denoting ρ the natural lift extension to states of a value abstraction $\rho \in uco(\mathbb{V})$. Consider a variable x in $\mathtt{P}_{[x]}$. Its values are approximated in ρ when evaluated in $[\![\cdot]\!]^{(\rho,\rho)}$.

DEFINITION 5.4. *The program $\mathtt{P}_{[x]} \in \mathbb{P}$ is ρ-stable w.r.t. the variable x if for any $d, z, y \in \mathbb{V}$:*

$$\rho(z) \neq \rho(y) \;\Rightarrow\; [\![\mathtt{P}_{[\rho(z)]}]\!]^{(\rho,\rho)}(d) = [\![\mathtt{P}_{[\rho(y)]}]\!]^{(\rho,\rho)}(d)$$

Instability is given by negation:

$$\exists z, y, d \in \mathbb{V}.\, \rho(z) \neq \rho(y) \;\wedge\; [\![\mathtt{P}_{[\rho(z)]}]\!]^{(\rho,\rho)}(d) \neq [\![\mathtt{P}_{[\rho(y)]}]\!]^{(\rho,\rho)}(d)$$

namely, there exist values for x that causes a variation in the output observation. The notion of stability corresponds to a slight weakening of a form of abstract non-interference, considering d as public, z and y as secret, but observing the whole output (and not only the public part). In particular, since we have a condition on an abstraction of the secret data, this corresponds to the so called *declassified non-interference via blocking* [20], supposing that the desired property is stability requires that no variations of the secret property ρ have to flow in the output observation. In our case the desired property is its negation, namely instability. In this case we aim that a variation of the secret property ρ induces a variation in the observable output. The notion of *flow-irredundancy* introduced in ANI [19] provides a stronger property than instability, avoiding existential quantification, which is problematic here.

DEFINITION 5.5. *A property ρ is said to be flow-irredundant wrt. a function f if $\forall z, y \in \mathbb{V}$:*

$$\rho(z) \neq \rho(y).\; \exists d \in \mathbb{V}.\, f(\rho(z), d) \neq f(\rho(y), d).$$

Considering as f the abstract interpretation of $\mathtt{P}$, we can observe that flow-irredundancy is strictly stronger than instability due to the initial universal quantifiers.

5.3 Modeling flow-irredundancy

Let us provide a set of rules for approximating the flow-irredundancy property of a program.

DEFINITION 5.6. *Let x, y be variables and $\mathtt{P}$ a program:*

- *$y \xrightarrow{\rho}_{\mathtt{P}_{[y]}} x$ (simply denoted $y \xrightarrow{\rho}_{\mathtt{P}} x$) iff*

$$\forall v_1, v_2 \in \rho\,.\,\exists \sigma \in \mathbb{D}\,.\,[\![\mathtt{P}_{[v_1]}]\!]^{(\rho,\rho)}\sigma \neq_x [\![\mathtt{P}_{[v_2]}]\!]^{(\rho,\rho)}\sigma$$

  where $\neq_x$ means that the result is compared only on x;
- *$y \overset{\rho}{\rightsquigarrow} e_{[y]}$ (simply denoted $y \overset{\rho}{\rightsquigarrow} e$), for an expression e, iff*

$$\forall v_1, v_2 \in \rho\,.\,\exists \sigma \in \mathbb{D}\,.\,[\![e_{[v_1]}]\!]^{(\rho,\rho)}\sigma \neq [\![e_{[v_2]}]\!]^{(\rho,\rho)}\sigma;$$

- *$y \overset{\rho}{\rightsquigarrow} \mathtt{P}_{[y]}$ (denoted $y \overset{\rho}{\rightsquigarrow} \mathtt{P}$) iff $\exists x \in Var(\mathtt{P})$ such that $y \xrightarrow{\rho}_{\mathtt{P}} x$.*

The set of all the variables modified by a program fragment is:

$$\textsc{Mod}(\mathtt{P}) \overset{\text{def}}{=} \left\{\; x \in Var(\mathtt{P}) \;\middle|\; \exists \sigma \in \mathbb{D}.\, [\![\mathtt{P}]\!]\sigma \neq_x \sigma \;\right\}$$

the set $\textsc{Mod}(\mathtt{P})^{(\rho,\rho)}$ is analogously defined, but instead of standard semantics we use an abstract one. Let us define the following rules:

$$\frac{y \xrightarrow{\rho}_{\mathtt{P}} x}{y \overset{\rho}{\rightsquigarrow} \mathtt{P}} \;\; \forall x \in Var(\mathtt{P}) \qquad \frac{y \overset{\rho}{\rightsquigarrow} e}{y \xrightarrow{\rho}_{x \,:=\, e} x}$$

$$\frac{y \overset{\rho}{\rightsquigarrow} \mathtt{C}_1,\; \exists x \in \textsc{Mod}^{(\rho,\rho)}(\mathtt{C}_1).\; y \xrightarrow{\rho}_{\mathtt{C}_1} x,\; x \overset{\rho}{\rightsquigarrow} \mathtt{C}_2}{y \overset{\rho}{\rightsquigarrow} \mathtt{C}_1 ; \mathtt{C}_2}$$

$$\frac{y \xrightarrow{\rho}_{\mathtt{C}_1} x,\; x \notin \textsc{Mod}^{(\rho,\rho)}(\mathtt{C}_2)}{y \xrightarrow{\rho}_{\mathtt{C}_1 ; \mathtt{C}_2} x} \qquad \frac{y \xrightarrow{\rho}_{\mathtt{C}_2} x}{y \xrightarrow{\rho}_{\mathtt{C}_1 ; \mathtt{C}_2} x}$$

$$\frac{y \overset{\rho}{\rightsquigarrow} \mathtt{C}}{y \overset{\rho}{\rightsquigarrow} \textbf{while } B \textbf{ do } \mathtt{C} \textbf{ endw}} \qquad \frac{y \overset{\rho}{\rightsquigarrow} B,\; x \in \textsc{Mod}^{(\rho,\rho)}(\mathtt{C})}{y \xrightarrow{\rho}_{\textbf{while } B \textbf{ do } \mathtt{C} \textbf{ endw}} x}$$

THEOREM 5.7. *The system of rules above is sound wrt. Definition* 5.6. *Moreover* $x \overset{\rho}{\rightsquigarrow} P$ *iff* ρ *is flow-irredundant wrt.* $[\![P]\!]^{(\rho,\rho)}$ *and therefore this implies* $P_{[x]}$ *is not* ρ*-stable wrt. the variable* x.

Consider the program $\mathtt{P}_{[\mathtt{P}_i]}$, where $\mathtt{P}_i$ are the program fragments to obfuscate, each of them computing values stored in variables $x_i \in \mathit{Var}(\mathtt{P})$. Suppose it is unstable, namely there exists $\rho(z_i) \neq \rho(y_i)$ that imply different observations. Hence, we consider a family of program fragments $\mathtt{Q}_i$ (distortions of $\mathtt{P}_i$) such that: (i) $[\![\mathtt{P}_i]\!] = [\![\mathtt{Q}_i]\!]$ (required by obfuscation); (ii) $\rho([\![\mathtt{P}_i]\!]) = [\![\mathtt{P}_i]\!]^{(\rho,\rho)}$ (we suppose the original program was complete, for this reason we need obfuscation); (iii) $\rho([\![\mathtt{Q}_i]\!]) \neq [\![\mathtt{Q}_i]\!]^{(\rho,\rho)}$ guaranteeing that the distorted program makes the observer lose precision. Hence, there exists two values of ρ, i.e., $[\![\mathtt{P}_i]\!]^{(\rho,\rho)}$ and $[\![\mathtt{Q}_i]\!]^{(\rho,\rho)}$ that induce instability, i.e., suppose $\mathtt{Q}_i$ are built in such a way that the abstract semantics of $\mathtt{Q}_i$ correspond to the values inducing instability.

PROPOSITION 5.8. *Let* $\rho \in \mathit{uco}(\mathbb{V})$ *and* $\mathtt{P}_{[\mathtt{P}_i]}$ *be a complete program for* (ρ,ρ)*, i.e.,* $\rho([\![\mathtt{P}]\!]) = [\![\mathtt{P}]\!]^{(\rho,\rho)}$ *and* ρ*-unstable. Consider* $\mathtt{P}_i$ *and* $\mathtt{Q}_i$ *defined as above, satisfying the conditions* (i)*,* (ii)*, and* (iii)*. Then the distorted program* $\mathtt{P}_{[\mathtt{Q}_i]}$ *is incomplete for* (ρ,ρ).

The next theorem gives a sufficient condition for systematically deriving obfuscated code by specializing interpreters. The idea is that, for an unstable program $\mathtt{P}$, the specialized code has to leave as residual the code for which the attacker is incomplete. This ensures that specialized code keeps incompleteness, becoming therefore obfuscated. Instability ensures that the resulting specialized interpreter includes incomplete structures as residual in the transformed program, making the resulting program obscure.

THEOREM 5.9. *Let* $\mathtt{P} \in \mathbb{P}$ *be an unstable program for given abstractions. Then there exists an interpreter* $\widetilde{\mathtt{Int}}$ *such that* $\mathfrak{O}(\mathtt{P}) = [\![\mathtt{spec}]\!](\widetilde{\mathtt{Int}}, \mathtt{P})$ *is an obfuscation in the sense of Theorem* 5.1.

The next section shows how the distorted interpreter could be designed for the case of opaque predicate insertion [9, 14, 15] and data-type refinement obfuscation [17]. In both cases the obfuscated code can be generated systematically by modifying an interpreter making incomplete structures residual in specialized code.

5.4 Opaque predicates

Opacity is an obfuscation technique based on the idea of confusing the control structure of a program by inserting predicates that are always true (or false) independently of the memory [9]. This means that the attacker deceived by opacity is exactly the one observing only the control structure of programs. In order to show that opacity is making incomplete an abstract interpreter, we characterize the construction of the control-flow graph (the attack) as an abstract interpretation and prove that a program contains opaque predicates if and only if this abstract interpretation is incomplete. The input abstraction ignores the evaluation of *any* control statement guard, considering always both the branches of computation; and the output abstraction ignores the memory (unnecessary when looking at the control structure of programs) and merges the collected histories of computations obtained. Note that, the first abstraction is indeed similar to the function $\mathcal{F}$ defined for flattening in Section 4.2, with the only difference that the branch information is *always* lost, and for this reason also this abstraction is specified in terms of $\mathcal{B}$ defined in Section 4.2, while the output abstraction is a further abstraction of the function $\mathcal{C}$ defined in Section 4.2.

Abstracting control predicates. The abstract interpretation here loses the evaluation of guarded commands such as **if** and **while**. The next function is a recursive map that *skips* the evaluation of the guards. The recursive function $\mathcal{B}^f$ decides when to skip the evaluation of the guard, while $\mathcal{B}$ (defined in Section 4.2) considers all the possible program lines that are reachable from the executed one, independently from the memory. Nested control structures require a recursive definition of abstraction:

$$\mathcal{B}^f(\langle\sigma,\langle l_1,l_2\rangle,\mathtt{G}\rangle) = \begin{cases} \mathcal{B}^f \circ \mathcal{B}(\langle\sigma,\langle l_1,l_2\rangle,\mathtt{G}\rangle) & \text{if } \mathtt{Stm}_{\mathtt{P}}(l_2) \in \{\textbf{if},\textbf{while}\} \\ \langle\sigma,\langle l_1,l_2\rangle,\mathtt{G}\rangle & \text{otherwise} \end{cases}$$

We abuse notation by denoting $\mathcal{B}^f$ also its additive lift and by using the notation: $\mathcal{B}^f(\langle\sigma,\langle l_1,l_2\rangle,\mathtt{G}\rangle) = \mathit{lfp}_{\langle\sigma,\langle l_1,l_2\rangle,\mathtt{G}\rangle}\mathcal{B}^f$.

PROPOSITION 5.10. $\mathcal{B}^f \in \mathit{uco}(\wp(\mathbb{D}))$.

Constructing CFG. We build the control-flow graph of an imperative program as the fix-point abstraction of the concrete semantics: $[\![\mathtt{P}]\!]_{\text{CFG}} = \mathit{lfp}\,(g_{\mathcal{L}}^{\text{CFG}})_{|\mathbb{G}}$ where $g_{\mathcal{L}}^{\text{CFG}} \overset{\text{def}}{=} \mathcal{C} \circ g_{\mathcal{L}} \circ \mathcal{B}^f$.

Opacity by incompleteness. Opaque predicates [7] are predicates whose values are known a priori, i.e. statically, and are independent from the state in which the predicate is evaluated.

DEFINITION 5.11 ($\mathcal{S}$-opaque predicate). *A predicate* B *is* $\mathcal{S}$*-opaque iff* $\forall\sigma \in \mathbb{M}.\ [\![B]\!]\sigma = \mathit{true}$ *or* $\forall\sigma \in \mathbb{M}.\ [\![B]\!]\sigma = \mathit{false}$.

In contrast with static opaque predicates, dynamic opaque predicates are predicates whose truth values depend on a dynamic property of the environment in which the predicate is evaluated. Therefore opacity depends on the property that the environment in which the predicates are evaluated will yield an always true or always false value (e.g, see opaque predicates from pointer-aliasing [7, 9]).

DEFINITION 5.12 ($\mathcal{D}$-opaque predicate).
Let $st \in \{\textbf{if}\ B\ \textbf{then}\ \mathtt{C}_1\ \textbf{else}\ \mathtt{C}_2, \textbf{while}\ B\ \textbf{do}\ \mathtt{C}\}$, $\mathtt{P} = \mathtt{P}'; st$, $S' = [\![\mathtt{P}']\!]S$, $S \subseteq \mathbb{D}$ *and* $S' = \langle M,\langle l_1,l_2\rangle,\mathtt{G}\rangle$ *with* $M \subseteq \mathbb{M}$. B *is* $\mathcal{D}$*-opaque w.r.t.* S *if* $\forall s \in S'.\ [\![B]\!]s = \mathit{true}$ *(denoted* $\mathcal{D}_T^S$*-opaque) or* $\forall s \in S'.\ [\![B]\!]s = \mathit{false}$ *(denoted* $\mathcal{D}_F^S$*-opaque).*
A predicate B *is* $\mathcal{D}$*-opaque iff* $\forall S \subseteq \mathbb{D}$. B *is either* $\mathcal{D}_T^S$*-opaque or* $\mathcal{D}_F^S$*-opaque.*

For instance the predicate $x^2 \geq 0$ is always true and it is a $\mathcal{S}$-opaque predicate. Instead, in $n := |n| + 2;$ **if** $n > 1$ **then** $\mathtt{C}$ ..., the predicate $n > 1$ is not $\mathcal{S}$-opaque, but it is $\mathcal{D}$-opaque. With program

$$n := |n| + 2; \textbf{while}\ n > 1\ \textbf{do}\ n := n - 1;$$

the predicate is no more $\mathcal{D}$-opaque, since there exists an iteration of the while, making the predicate false. Indeed, in presence of loops, only guards of a non terminating while can be $\mathcal{S}$/$\mathcal{D}$-opaque.

EXAMPLE 5.13. *Consider* $\Pi = \textbf{if}\ B\ \textbf{then}\ \mathtt{C}_1\ \textbf{else}\ \mathtt{C}_2\ \textbf{fi}$ *(*l_Π *the initial program line of* Π*,* l_1 *the program line of* $\mathtt{C}_1$*,* l_2 *the program line of* $\mathtt{C}_2$*, and* l_e *the exit point).* $\mathtt{G}$ *is s.t.* $\text{Nodes}(\mathtt{G}) = \{l_\Pi, l_1, l_2, l_e\}$ *(for simplicity we omit the edges). Consider* $\mathcal{C} \circ g_{\mathcal{L}} \circ \mathcal{B}^f(\Pi)$*: if* B *is not opaque* $\mathcal{C} \circ g_{\mathcal{L}}(\Pi)$ *generates* $\mathtt{G}'$ *such that* $\text{Nodes}(\mathtt{G}') = \{l_\Pi, l_1, l_2\}$*, while if* B *is (statically) opaque (for instance always true) we obtain a graph* $\mathtt{G}''$ *such that* $\text{Nodes}(\mathtt{G}'') = \{l_\Pi, l_1\}$*. Note that in both cases we have incompleteness (*$\mathtt{G} \neq \mathtt{G}'$ *and* $\mathtt{G} \neq \mathtt{G}''$*), but while the difference between* $\mathtt{G}$ *and* $\mathtt{G}'$ *is only* l_e*, that will be added by* $g_{\mathcal{L}}$ *in the fix-point computation, the difference between* $\mathtt{G}$ *and* $\mathtt{G}''$ *is in the presence of* l_2 *that the fix-point computation of* $g_{\mathcal{L}}$ *cannot add, being* B *always true. Suppose* B *is* $\mathcal{D}$*-opaque w.r.t. some set of memories* $M_B \subseteq \mathbb{M}$*, then for any* $\mathtt{P}_1$ *such that there exists* $M \subseteq \mathbb{M}$ *s.t.* $[\![\mathtt{P}_1]\!](M) \not\subseteq M_B$*, we have that when computing* $\mathtt{P}_1; \Pi$*,* B *is no more opaque, and therefore we lose the incompleteness property on* $\mathtt{P}_1; \Pi$*, at least for* Π.

LEMMA 5.14. *Let* $S = \langle M,\langle l_1,l_2\rangle,\mathtt{G}\rangle \subseteq \mathbb{D}$ *with* $M \subseteq \mathbb{M}$.

1. $\forall n \in \mathbb{N}.\ (\mathcal{C} \circ g_{\mathcal{L}})^n(S) \sqsubseteq_{\mathbb{G}} (g_{\mathcal{L}}^{CFG})^n(S)$
2. $\forall n \in \mathbb{N}.\ \mathcal{C} \circ g_{\mathcal{L}}^n(S) \sqsubseteq_{\mathbb{G}} (g_{\mathcal{L}}^{CFG})^n(S)$

The next theorem characterizes static and dynamic opacity in terms of completeness of abstract interpretation.

THEOREM 5.15. *Let* $\mathtt{P} \in \mathbb{P}$.

1. $[\![\mathtt{P}]\!]^{\mathcal{C}}_{\mathbb{G}} =_{\mathbb{G}} [\![\mathtt{P}]\!]_{CFG}$ *iff* P *doesn't contain* $\mathcal{S}$*-opaque predicates;*
2. $\mathcal{C}([\![\mathtt{P}]\!]_{\mathbb{G}}) =_{\mathbb{G}} [\![\mathtt{P}]\!]_{CFG}$ *iff* P *doesn't contain* $\mathcal{D}$*-opaque predicates.*

Because in general $\mathcal{C}([\![\mathtt{P}]\!]_{\mathbb{G}}) \sqsubseteq_{\mathbb{G}} [\![\mathtt{P}]\!]^{\mathcal{C}}_{\mathbb{G}}$, as expected, the absence of $\mathcal{S}$-opacity implies the absence of $\mathcal{D}$-opacity. Moreover the latter can only be ensured by enforcing completeness relatively to a concrete enough semantics that dynamically executes the programs, such as $[\![\mathtt{P}]\!]_{\mathbb{G}}$, while in the static case completeness is related with a more abstract semantics that loses the memory associated with each state, making all predicates equivalently evaluated to an unknown (including either true or false) value.

COROLLARY 5.16. *If* $\mathfrak{O} : \mathbb{P} \to \mathbb{P}$ *is a program transformation adding only* $\mathcal{S}$*-opaque predicates to a program* P *(not containing opaque predicates), then* $\mathcal{C}([\![\mathfrak{O}(\mathtt{P})]\!]_{\mathbb{G}}) = [\![\mathfrak{O}(\mathtt{P})]\!]^{\mathcal{C}}_{\mathbb{G}}$.

These results tell us that, while for $\mathcal{S}$-opacity we can easily design a distorted interpreter that simply selects and inserts, in the interpretation loop, predicates from a database of $\mathcal{S}$-opaque predicates, this is not in general possible for $\mathcal{D}$-opacity. The intuition beyond this negative result is that $\mathcal{D}$-opacity depends on the semantics of the program containing the predicate, i.e., it is the semantics (viz., the computed environment) of the previous statements that makes the predicate $\mathcal{D}$-opaque. While instead $\mathcal{S}$-opacity is a property independent on the surrounding code.

5.5 Data-type obfuscation

Let us consider the obfuscation techniques based on the encoding of data [17]. In this case obfuscation is achieved by data-refinement, namely by exploiting the complexity of more complex data-structures or values in such a way that actual computations can be viewed as abstractions of the refined (obfuscated) ones. The idea consists in choosing a pair of statements c^α and c^γ such that $c^\gamma; c^\alpha \equiv$ **skip**. This means that both c^α and c^γ are statements of the form: $c^\alpha \equiv x := G(x)$ and $c^\gamma \equiv x := F(x)$, for some function F and G. A program transformation $\mathfrak{O}(\mathtt{P}) \stackrel{\text{def}}{=} c^\gamma; \tau_x(\mathtt{P}); c^\alpha$ is data-type obfuscation for data-type x if $\mathfrak{O}(\mathtt{P}) \equiv \mathtt{P}$, where τ_x adjusts the data-type computation for x on the refined type (see [17]). It is known that data-type obfuscation can be modeled as adjoint functions (Galois connections), where c^γ represents the program concretizing , viz. refining, the datum x and c^α represents the program abstracting the refined datum x back to the original data-type. As proved in [18], this is precisely modeled as a pair of adjoint functions: $\alpha : \mathbb{V} \longrightarrow \mathbb{V}^{\Re}$ and $\gamma : \mathbb{V}^{\Re} \longrightarrow \mathbb{V}$ relating the standard data-type $\mathbb{V}$ for x with its refined version $\mathbb{V}^{\Re}$.

EXAMPLE 5.17. *Consider* $\mathtt{P} = \ x := x + 2;$*,* $c^\alpha \equiv x := x/2$ *and* $c^\gamma \equiv x := 2x$*, then we have* $\tau_x(\mathtt{P}) = \ x := 2(x/2 + 2)$*, namely* $x := x + 4$*, therefore:* $\mathfrak{O}(\mathtt{P}) \equiv x := 2x; x := x + 4; x := x/2$*. Consider, for instance a more complex program:*

$\mathtt{P} = x := 1; s := 0;$ **while** $x < 15$ **do** $s := s+x; x := x+1;$ **endw**

Then we have

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

α*,* γ*,* $\mathbb{V}$*, and* $\mathbb{V}^{\Re}$ *are here the most obvious ones.*

THEOREM 5.18. *Let* $\rho \in$ *uco*$(\mathbb{V})$*. If* $x \stackrel{\rho}{\rightsquigarrow} P_{[x]}$ *then for any pair of adjoint functions* (α, γ) *such that* $\gamma\alpha \sqsubseteq \rho$ *and program refinement* τ_x *mapping programs into incomplete structures of* $\mathcal{L}$ *for* ρ*:* $\rho([\![\mathtt{P}_{[\mathfrak{O}(\mathtt{Q})]}]\!] \sqsubset [\![\mathtt{P}_{[\mathfrak{O}(\mathtt{Q})]}]\!]^{(\rho,\rho)}$.

Data-type obfuscation can be easily implemented by specializing an interpreter, straight from Theorem 5.9.

EXAMPLE 5.19. *A very simple data obfuscation can be generated as specialization of the simple self-interpreter of Example* 3.1*. In this case, it is not control, but data that are obfuscated. The technique is similar to that of Drape* et al *[16, 27], but automated by interpreter specialization. In this example the simple self-interpreter of Example* 3.1*, is modified so that all values in the store are obfuscated by the simple trick of adding* 1*. Mutual inverse functions* $obf(x)$ *and* $dob(x)$ *respectively obfuscate or invert the obfuscation. Input values are obfuscated in the initial store, and output values are de-obfuscated in the program's final store. Expression evaluation obfuscates constants. Also, it first de-obfuscates subexpression values, then applies the desired operation (here* $+$ *or* $-$*), and then obfuscates the result.*

```
input P, d;                                   Program to be interpreted, and its data
pc := 2; store := [in ↦ obf(d), out ↦ obf(0), x1 ↦ obf(0), ...];
while pc < length(P) do
    instruction := lookup(P, pc);
    case instruction of                                   Dispatch on syntax
    skip     : pc := pc + 1;
    x := e   : store := store[x ↦ eval(e, store)]; pc := pc + 1;
    ...  endw ;
output dob(store[out]);
obf(x) = x + 1; dob(x) = x − 1            Obfuscation/de-obfuscation
eval(e, store) = case e of
    constant : obf(e)                                     Obfuscate constants
    variable : store(e)
    e1 + e2  : obf(dob(eval(e1, store)) + dob(eval(e2, store)))
    e1 − e2  : obf(dob(eval(e1, store)) − dob(eval(e2, store)))
    ...
end
```

The program in Example 3.1 *is automatically transformed into an equivalent program with obfuscated expressions and data:*

```
1. input x;
1.5. x := x + 1;
2. y := 2 + 1;
3. while x − 1 > 0 + 1 − 1 do
       4. y := ((y − 1) + (2 + 1 − 1)) + 1;
       5. x := (x − (1 + 1 − 1)) + 1 endw
6. output y − 1;
7. end
```

A wide variety of data obfuscations can be performed by modifying the interpreter `interp`. For example, variables could be represented in pairs: (x, y) could be uniquely and decipherably be represented by $(x + y, x - y)$. Realizing this in practice involves changing the `interp` code when variables are fetched (expression x) and stored ($x := e$). While this would make $\mathtt{P}'$ run somewhat slower than P, it would be only by a constant factor and not asymptotically slower. On the other hand, familiar abstract interpretations such as live variables, constant propagation, available expressions, etc. would be substantially hindered by such a representation.

6. Discussion

The most related papers are [18] and [29]. In [18] the authors introduced the idea of information hiding by making an abstract interpretation incomplete. The corresponding transformations were ad hoc due to the fact that the basic code transformers (which exist under restrictive hypothesis) were not semantics preserving. In [29] the authors introduced the notion of *obfuscated interpretation*, for hiding functionality of a given program. The idea is to reduce the problem of catching the hidden functionality to the problem of knowing the way the obfuscated interpreter implements instruc-

tions. This is an instance of our approach, where the attacker is an abstract interpreter that is blind to context-dependent semantics.

Real-world attacks typically use combined attack strategies, each one employing a fixed number of analysis tools and methods. We believe that these can be formalized as the combination of abstract interpreters which can be defeated by specializing corresponding distorted interpreters. In principle, the combination of these distorted interpreters may lead to a potentially robust obfuscation against an arbitrary combination of attacks. Future work concerns the development and combination of protection mechanisms based on specialization of distorted interpreters for execution flow integrity, function boundary concealment, and defense against code decompilation and program tracing. The modularity of our approach has here several advantages. For instance, Futamura projections (e.g., [24]) are as follow for distorted interpreters $\texttt{interp}^{+}$:

1.	$\texttt{P}'$	:=	$[\![\texttt{spec}]\!](\texttt{interp}^{+}, \texttt{P})$	Transform program
2.	comp	:=	$[\![\texttt{spec}]\!](\texttt{spec}, \texttt{interp}^{+})$	Generate transformer
3.	cogen	:=	$[\![\texttt{spec}]\!](\texttt{spec}, \texttt{spec})$	Transformer generator

The *obfuscating transformations* we have seen are clearly instances of the 1st projection. An *obfuscating compiler* has been generated by the 2nd projection using UNMIX. For example, if P is $\texttt{interp}^{\texttt{flat}}$, then comp is a *stand-alone "flattening" obfuscator*. Immediate consequences are other better ways to transform and to produce a transformer: $\texttt{P}' = [\![\texttt{comp}]\!](\texttt{P})$ (transform by compiler) and $\texttt{comp} = [\![\texttt{cogen}]\!](\texttt{interp}^{+})$ (generate transformer). Future developments will involve gaining a deeper understanding in expected slowdowns, namely the relations between: (a) $time_{\texttt{P}'}(\texttt{d})$ and $time_{\texttt{P}}(\texttt{d})$, exemplifying the slowdown imposed by the flattening obfuscation; (b) $time_{\texttt{spec}}(\texttt{interp}^{\texttt{flat}}, \texttt{P})$ and $length(\texttt{P})$, exemplifying the amount of time required to do the flattening obfuscation by general specialization; (c) $time_{\texttt{comp}}(\texttt{P})$ and $length(\texttt{P})$, exemplifying the amount of time required to do the flattening obfuscation by running the UNMIX-generated program obfuscator.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *26th ACM POPL '99*, pp. 147–160. 1999.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01*, LNCS 2139, pp. 1–18. 2001.

[3] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *ACM PLDI '93*, pp. 46–55, 1993.

[4] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *ISC '01*, LNCS 2200, pp. 144–155. 2001.

[5] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A white-box des implementation for drm applications. In *Digital Rights Management Workshop*, LNCS 2696, pp. 1–15. 2003.

[6] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[7] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009. ISBN 0321549252.

[8] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obduscation-tools for software protection. *IEEE Trans. Software Eng.*, pp. 735–746, 2002.

[9] C. Collberg, C. D. Thomborson, and D. Low. Manufactoring cheap, resilient, and stealthy opaque constructs. In *25st ACM POPL '98*, pp. 184–196. 1998.

[10] C. Consel, J. Lawall, and A.-F. L. Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52 (17(1)):47–92, 2004.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM POPL '77*, pp. 238–252. 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM POPL '79*, pp. 269–282. 1979.

[13] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th ACM POPL '02*, pp. 178–190. 2002.

[14] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. *J. of Comp. Security*, 17(6):855–908, 2009.

[15] M. Dalla Preda, M. Madou, K. D. Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. In *11th AMAST '06*, LNCS 4019, pp. 81–95. 2006.

[16] S. Drape. *Obfuscation of Abstract Data-Types*. PhD thesis, University of Oxford, 2004.

[17] S. Drape, C. Thomborson, and A. Majumdar. Specifying imperative data obfuscations. In *ISC'07*, LNCS 4779, pp. 299–314. 2007.

[18] R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *6th IEEE SEFM'08*, pp. 7–20. 2008.

[19] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *31st ACM POPL '04*, pp. 186–197. 2004.

[20] R. Giacobazzi and I. Mastroeni. Adjoining classified and unclassified information by abstract interpretation. *J. of Computer Security.*, 18(5):751-797. 2010.

[21] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In *8th SAS'01*, LNCS 2126, pp. 356–373. 2001.

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

[23] N. D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52(17(1)):307–339, 2004.

[24] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[25] J. Jørgensen. Similix: A self-applicable partial evaluator for scheme. In *Partial Evaluation*, LNCS 1706, pp. 83–107. 1998.

[26] M. Leuschel. Advanced logic program specialisation. In *Partial Evaluation - Practice and Theory, DIKU 1998 Int. Summer School*, pp. 271–292. 1999. ISBN 3-540-66710-5.

[27] A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *DRM '07*, pp. 70–81. ACM, 2007.

[28] I. Mastroeni and D. Zanardini. Data dependencies and program slicing: From syntax to abstract semantics. In *ACM PEPM'08*, pp. 125 – 134. 2008.

[29] A. Monden, A. Monsifrot, and C. D. Thomborson. A framework for obfuscated interpretation. In *ACSW Frontiers, AISW2004*, pp. 7–16, 2004.

[30] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding – A survey. *Proc. of the IEEE*, 87(7):1062–1078, 1999.

[31] T. Reps and W. Yang. The semantics of program slicing and program integration. In *Colloq. on Current Issues in Programming Languages*, LNCS 352, pp. 360–374. 1989.

[32] S. Romanenko. *Unmix, a specializer for a subset of Scheme:* `http://code.google.com/p/unmix/` . Keldysh Institute, Moscow, 1993-2009.

[33] P. Thiemann. Aspects of the PGG system: Specialization for standard scheme. In *Partial Evaluation - Practice and Theory, DIKU 1998 Int. Summer School*, pp. 412–432. Springer, 1999. ISBN 3-540-66710-5.

[34] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *IEEE International Conference of Dependable Systems and Networks*, pp. 193–202, 2001.

[35] M. Ward and H. Zedan. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.