Obfuscation by Partial Evaluation of Distorted Interpreters

Neil D. Jones
DIKU, University of Copenhagen (prof. emeritus)

Joint work with Roberto Giacobazzi and Isabella Mastroeni
University of Verona

Paper: in PEPM 2012

WHAT AND WHY ?

Obfuscation = hiding of intended meaning in communication, making a program confusing, wilfully ambiguous, and harder to interpret.

- Making something dark
- Putting something in a shadow

Our version: a semantics-preserving program transformation intended to make transformed programs hard to understand.

Popular in the computer security and software engineering communities: Obfuscation, Watermarking, Steganography. WHY ?

- To avoid theft of an algorithm (hard to steal or adapt or patent)
- to hide evidence in a program of its authorship, ownership, creator
- . . . related other security techniques . . .

OBFUSCATION = A PROGRAM TRANSFORMATION $\mathrm{p} \mapsto \mathrm{p}'$

A scenario:

- An attacker is trying to analyze or decipher obfuscated program p'
- a defender is trying to construct p' to make this hard to do

Want: p' is executable, but it should be hard to adapt, exploit, or analyze.

SOME CRITERIA:

1. Semantics preservation: we must have

$$\forall \mathrm{p} \,.\, [\![\mathrm{p}']\!] = [\![\mathrm{p}]\!]$$

2. *Automation*: p' is obtained from p without hand work.

   Thus the programmer/inventor of p can release p' instead of p.

3. *Efficiency*: p' should not be too much slower or larger than p.
4. *Potency* = hard reverse engineering, namely

   p is hard to obtain from its obfuscated version p'.

HOW ? OUR SILVER BULLET

Program transformation by **specializing a self-interpreter**.

Program `interp` is a *self-interpreter* if for all programs `p` and data $\mathtt{d} \in \mathbb{D}$

$$[\![\mathtt{p}]\!](\mathtt{d}) = [\![\mathtt{interp}]\!](\mathtt{p}, \mathtt{d})$$

A *partial evaluator* (or *program specializer*) `spec` satisifies for every program `p` with "static" input $\mathtt{s} \in \mathbb{D}$ and "dynamic" input $\mathtt{d} \in \mathbb{D}$, that

$$[\![\mathtt{p}]\!](\mathtt{s}, \mathtt{d}) = [\![\ [\![\mathtt{spec}]\!]\,\mathtt{(p, s)} \ ]\!]\,(\mathtt{d})$$

Some practical program specializers:
TEMPO and CMIX for C; ECCE and LOGEN for Prolog; UNMIX, SIMILIX and PGG for SCHEME.

We used Unmix in our experiments.

PROGRAM TRANSFORMATION BY INTERPRETER SPECIALIZATION

Suppose

$$\mathtt{p}' := [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{p})$$

So $\mathtt{p}'$ is the result of specializing a self-interpreter to program $\mathtt{p}$.

Claim: $[\![\mathtt{p}]\!] = [\![\mathtt{p}']\!]$, by simple equational reasoning. For any data $\mathtt{d}$,

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

Therefore the function

$$\mathtt{p} \mapsto [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{p})$$

is a *semantics-preserving program transformer*.

OPTIMAL SPECIALIZATION VERSUS OBFUSCATION

In the transformation

$$\mathtt{p} \mapsto \mathtt{p}' = [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{p})$$

For optimal specialization:

$\mathtt{p}'$ **should be as efficient as** $\mathtt{p}$**.**

But... for good obfuscation:

$\mathtt{p}'$ **should be harder to understand than** $\mathtt{p}$**.**

Conflicting goals, but achievable by (re-)designing `interp` cleverly.

We show that several useful program obfuscations can be obtained by interpreter specialization.

A bit of useful slack:

> **It is OK for `interp` to be slow, as long as**
> $\mathtt{p}'$ **is fast enough, and hard enough to understand.**

IN GENERAL, IF $\mathtt{p}' = [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{p})$:

1. Program $\mathtt{p}'$ inherits the ***algorithm*** of program $\mathtt{p}$.
2. Program $\mathtt{p}'$ inherits the ***programming style*** of `interp`.

Our trick: build a *program transformer*

- by programming a self-interpreter $\mathtt{interp}^+$
- in a style to give the desired transformation.
- Then (automatically)
  - specialise $\mathtt{interp}^+$ to any input program $\mathtt{p}$
  - to transform $\mathtt{p}$ as desired.

Some writing styles that can be inherited from `interp`:

- functional language, tail-recursive ,or CPS (continuation-passing) styles
- Or `interp` can use memoisation to implement function calls.

STRUCTURE OF A SIMPLE SELF-INTERPRETER

input p, d;	Program to be interpreted, and *its* data
$pc := 2$;	Initialise program counter
$store := [in \mapsto d, out \mapsto 0, x_1 \mapsto 0, \ldots]$;	Initialise store
while $pc < length(\mathtt{p})$ **do**	
$instruction := lookup(\mathtt{p}, \mathtt{pc})$;	Find the pc-th instruction
case $instruction$ **of**	Dispatch on syntax
skip : $pc := pc + 1$;	Once case per command type
$x := e$: $store := store[x \mapsto eval(e, store)]$; $pc := pc + 1$;	
... **endw** ;	
output $store[out]$;	
$eval(e, store) =$ **case** e **of**	Function to evaluate expressions
$constant : e$	
$variable : store(e)$	
$e1 + e2 : eval(e1, store) + eval(e2, store)$	
$e1 - e2 : eval(e1, store) - eval(e2, store)$	
$e1 * e2 : eval(e1, store) * eval(e2, store)$	
...	

SPECIALIZATION OF THE SIMPLE SELF-INTERPRETER

A simple color-coding scheme:

- **GREEN for static(ally computable)**
- **RED for dynamic.**

First steps using this coding:

- `interp` **variable** `p` **is classified as "static", and variable** d **is classified as "dynamic".**
- **. . . and a bit more:**
-
- `interp` **variables** e, pc **and** $instruction$ **are classified as "static", since given any** `p` **they can only assume finitely many values.**

STRUCTURE OF A SIMPLE SELF-INTERPRETER

```
input p, d;                                      Program to be interpreted, and its data
pc:=2;                                           Initialise program counter
store :=[in ↦ d, out ↦ 0, x1 ↦ 0, ...];           Initialise store
 while pc<length(p) do
     instruction:=lookup(p,pc);                  Find the pc-th instruction
     case instruction of                         Dispatch on syntax
     skip    : pc:=pc+1;                         Once case per command type
     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)
     ...
```

SPECIALIZATION OF THE SIMPLE SELF-INTERPRETER

- `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.**
-
- **The interpreter's while loop is unfolded, so the only remaining control transfers implement the transfers in program** `p`**.**

STRUCTURE OF A SIMPLE SELF-INTERPRETER

```
input p, d;                                   Program to be interpreted, and its data
pc:=2;                                        Initialise program counter
store :=[in ↦ d, out ↦ 0, x1 ↦ 0, ...];      Initialise store
 while pc<length(p) do
     instruction:=lookup(p,pc);               Find the pc-th instruction
     case instruction of                      Dispatch on syntax
     skip   : pc:=pc+1;                       Once case per command type
     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)
     ...
```

SPECIALIZATION OF THE SIMPLE SELF-INTERPRETER

- `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.
-
- The interpreter's **while** loop is unfolded, so the only remaining control transfers correspond to those present in program `p`.
-
- `interp` function $eval$ is completely unfolded and so does not appear in $\mathtt{p}'$, since all recursive calls decrease the static value e.

STRUCTURE OF A SIMPLE SELF-INTERPRETER

```
input p, d;                                          Program to be interpreted, and its data
pc:=2;                                               Initialise program counter
store :=[in ↦ d, out ↦ 0, x1 ↦ 0, ...];              Initialise store
 while pc<length(p) do
      instruction:=lookup(p,pc);                     Find the pc-th instruction
      case instruction  of                           Dispatch on syntax
      skip     :  pc:=pc+1;                          Once case per command type
      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)
      ...
```

SPECIALIZATION OF THE SIMPLE SELF-INTERPRETER

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

SPECIALIZATION OF THE SIMPLE SELF-INTERPRETER

Net effect of all of these tricks:

– the specialized program $\mathtt{p}' = [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{p})$

– is identical to $\mathtt{p}$ **(up to variable renaming).**

Alas, this is not what we want for obfuscation

since $\mathtt{p}'$ **is** α**-equivalent to** $\mathtt{p}$
(identical up to renaming $=$ **too easy to deobfuscate. . .)**

But it is a first step towards

more interesting automatic program transformation

AN EASY EXAMPLE: DATA OBFUSCATION

Data obfuscation **Similar to Drape's technique, but automated.**

Modify the simple self-interpreter so that

- **all values in the store are obfuscated, e.g., by multiplying by 2.**
- **Mutual inverse functions $obf(x)$ and $dob(x)$.**

Modify `interp` so that:

- **All stored values are obfuscated;**
  - **Input values are obfuscated in the initial store;**
  - **variable values are obfuscated just before putting in the store; and**
  - **output values are de-obfuscated in the program's final store.**
- **Expression evaluation yields obfuscated values:**

A VERY SIMPLE DATA OBFUSCATION: $V \mapsto 2V$

input $\mathtt{p}, \mathtt{d}$; **Program to be interpreted, and *its* data**

$pc := 2$; **Initialise program counter and obfuscated store:**

$store := [in \mapsto \textsf{obf}(d), out \mapsto \textsf{obf}(0), x_1 \mapsto \textsf{obf}(0), \ldots]$;

while $pc < length(\mathtt{p})$ **do**

$\quad instruction := lookup(\mathtt{p}, \mathtt{pc})$;

$\quad$ **case** $instruction$ **of** **Dispatch on syntax**

$\quad$ **skip** $: pc := pc + 1$; **Obfuscate values when stored:**

$\quad$ **x:=e** $: store := store[x \mapsto \textsf{obf}(eval(e, store))]; \; pc := pc + 1$;

$\quad \ldots$ **endw** ;

output $\textsf{dob}(store[out])$;

$\textsf{obf}(V) = 2 * V$; $\textsf{dob}(V) = V/2$ **Obfuscation/de-obfuscation**

$eval(e, store) =$ **case** e **of**

$\quad$ **constant** : $\textsf{obf}(e)$

$\quad$ **variable** : $store(e)$ **Variable values**

$\quad$ **e1 + e2** : $\textsf{obf}(\textsf{dob}(eval(e1, store)) + \textsf{dob}(eval(e2, store)))$

$\quad$ **e1 - e2** : $\textsf{obf}(\textsf{dob}(eval(e1, store)) - \textsf{dob}(eval(e2, store)))$

EXAMPLE OUTPUT FROM DATA OBFUSCATION

Program p

$^{1.}$**input** x;
$^{2.}$$y := 2$;
$^{3.}$**while** $x > 0$ **do**
  $^{4.}$$y := y + 2$;
  $^{5.}$$x := x - 1$
endw
$^{6.}$**output** y;
$^{7.}$**end**

is automatically transformed into this equivalent obfuscated program p':

$^{1.}$**input** x;
$^{1.5.}$$x := 2 * x$; Obfuscate input x
$^{2.}$$y := 2 * 2$; Obfuscate $y := 2$
$^{3.}$**while** $x/2 > 0$ **do** De-obfuscate x
  $^{4.}$$y := 2 * (y/2 + 2)$;
  $^{5.}$$x := 2 * (x/2 - 1)$
endw
$^{6.}$**output** $y/2$; De-obfuscate output
$^{7.}$**end**

WHAT IS HAPPENING ? HOW TO GENERALISE ?

1. **The above example** applies one-to-one functions

$$Value \underset{\lambda x\,.\,x/2}{\overset{\lambda x\,.\,2x}{\rightleftarrows}} Value$$

2. **More generally**: apply one-to-one functions

$$Value \underset{dob}{\overset{obf}{\rightleftarrows}} Value$$

3. **Still more generally**: apply one-to-one **store transformations**:

$$Store \underset{dob}{\overset{obf}{\rightleftarrows}} Store$$

A nasty example:

$$obf(x, y) = (x + y, x - y)$$

$$dob(u, v) = ((u + v)/2, (u - v)/2)$$

This mixes up values of different variables!

OBFUSCATION: $(X, Y) \mapsto (U, V) = (X + Y, X - Y)$

Program p

1. **input** x;
2. $y := 2$;
3. **while** $x > 0$ **do**
   4. $y := y + 2$;
   5. $x := x - 1$

   endw
6. **output** y;
7. **end**

is automatically transformed into this equivalent obfuscated program p'**:**

1. **input** x; $u := x + y$; $v := x - y$;	Initialise
2.1. $u := (u + v)/2 + 2$;	Obfuscated $y := 2$
2.2. $v := (u + v)/2 - 2$	-continued-
3. **while** $(u + v)/2 > 0$ **do**	De-obfuscated $x > 0$
4. $u := u + 2$; $v := v - 2$;	
5. $u := u - 1$; $v := v - 1$;	
endw	
6. **output** $(u - v)/2$;	De-obfuscate output
7. **end**	

OBFUSCATION FROM A "WHOLE-PROGRAM" VIEWPOINT

A conflict that makes program obfuscation a subtle problem 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
***should* not *be easy to understand*.**

This suggests (among other things):

obfuscation by making the program's control flow hard to understand.

EXAMPLE OF CODE FLATTENING

The following flattened program p' has

- **only one loop** (regardless of how many loops p has), and
- an **explicit program counter** pc

What has been obfuscated? **The flow of control!**

Original program p:

$^{1\cdot}$**input** x;
$^{2\cdot}y := 2$;
$^{3\cdot}$**while** $x > 0$ **do**
    $^{4\cdot}y := y + 2$;
    $^{5\cdot}x := x - 1$
  endw
$^{6\cdot}$**output** y;
$^{7\cdot}$**end**

Flattened equivalent program p':

$^{1\cdot}$**input** x; $^{2\cdot}pc := 2$;
$^{3\cdot}$**while** $pc < 6$ **do**
    $^{4\cdot}$**case** pc **of**
        $2 :$ $^{5\cdot}y := 2$; $^{6\cdot}pc := 3$;
        $3 :$ $^{7\cdot}$**if** $x > 0$ **then** $^{8\cdot}pc := 4$ **else** $^{9\cdot}pc := 6$;
        $4 :$ $^{10\cdot}y := y + 2$; $^{11\cdot}pc := 5$;
        $5 :$ $^{12\cdot}x := x - 1$; $^{13\cdot}pc := 3$;
  endw
$^{14\cdot}$**output** y
$^{15\cdot}$**end**

SPECIALIZATION OF THE "FLATTENING" INTERPRETER:

Essential trick: to recode `interp` so that the specializer will classify variable pc as *dynamic*. Technically

- pc **is made dynamic using the Unmix `generalize` annotation.**
- `interp` **is extended to have both dynamic and static copies of pc, so specialization will generate $\mathtt{p}'$ code such as**

$$\textbf{case} \ldots\; pc = 5 : \; x := x - 1; pc := 3$$

- **Since pc is dynamic, the while loop in $\mathtt{interp}^{\mathtt{flat}}$ will not be unfolded, and so pc comes to appear in the specialized program $\mathtt{p}'$.**

The transformation

$$\mathtt{p} \mapsto \mathtt{p}' = [\![\mathtt{spec}]\!](\mathtt{interp}^{\mathtt{flat}}, \mathtt{p})$$

will flatten *any* program; i.e., it is in no way specific to the example program $\mathtt{p}$ above.

WHAT IS "HARD ENOUGH" ?

- Ideally, the *complexity* of obtaining $\mathtt{p}$ from $\mathtt{p}'$ should be high (for example, NP-hard).

  View: the attacker is an arbitrary PTIME program.

  We don't know how to do this (one-way functions...).

  So we settle for an easier solution:

- Obfuscate so that $\mathtt{p}'$ is hard to abstractly interpret.

  View: the attacker is a program analyzer, e.g., as used in a compiler.

  We know better how to do this:

  - Use the concept of *complete abstract interpretation*

    (studied by Giacobazzi et al)

  - Design self-interpreter `interp` so that the abstract interpretation of $\mathtt{p}'$ is not complete (whether or not analysis of $\mathtt{p}$ would be complete)

SIGN ANALYSIS: COMPLETENESS AND INCOMPLETENESS

Abstract lattice for sign analysis: $\{\bot, +, 0, -, \top\}$

Sign analysis is complete for multiplication $*$: exact analysis information.

$*$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

Sign analysis is incomplete for addition $+$: imprecise analysis inform'n.

$+$	$-$	0	$+$
$-$	$-$	$-$	$\top(!)$
0	$-$	0	$+$
$+$	$\top(!)$	$+$	$+$

Our trick: let the interpreter evaluate $*$ using $+$

$$
\begin{array}{lll}
eval(e, store) & = & \textbf{case } e \textbf{ of} \\
\quad \texttt{e1 + e2} & : & eval(e1, store) + eval(e2, store) \\
\quad \texttt{e1 * e2} & : & \textbf{let } v1 = eval(e1, store), v2 = eval(e2, store) \\
 & & \textbf{in } \ v1 * (v2 - 1) + v1
\end{array}
$$

The $+$ makes analysis imprecise!

OBFUSCATION BY EXPLOITING INCOMPLETENESS

Program $\mathtt{p}$ **Sign analysis of** $\mathtt{p}$ **yields** $y \mapsto +$ **in**

input x;
$y := 2$;
while $x > 0$ **do**
    $y := \boxed{y * y}$;
    $x := x - 1$
    endw
output y;
end

is automatically transformed into this equivalent obfuscated program $\mathtt{p}'$**:**

input x;
$y := 2$;
while $x > 0$ **do**
    $y := \boxed{y * (y - 1) + y}$;
    $x := x - 1$
    endw
output y;
end

Sign analysis of $\mathtt{p}'$ **yields** $y \mapsto \top$**.**

EXAMPLE: ATTACK MODEL FOILED BY FLATTENING

In the paper, we show the control flow flattening-based obfuscation can be modeled as making incomplete an abstract interpretation.

The attacker is an abstract interpretation extracting the program's control flow graph.
CFG extraction be done several 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).
- An enriched concrete semantics yields a flow chart modeling the history of the computation.
- An abstraction of this concrete semantics yields a flow chart for any program.

Result: this (natural) abstract interpretation is incomplete for a flattened program.

META-LEVEL DISCUSSION

The Futamura projections are as follow for a distorted interpreter $\mathtt{interp}^+$**:**

1. $\mathtt{p}' := [\![\mathtt{spec}]\!](\mathtt{interp}^+, \mathtt{p})$ **Obfuscate one program**
2. $\mathtt{comp} := [\![\mathtt{spec}]\!](\mathtt{spec}, \mathtt{interp}^+)$ **Generate a stand-alone obfuscator**
3. $\mathtt{cogen} := [\![\mathtt{spec}]\!](\mathtt{spec}, \mathtt{spec})$ **A generator of obfuscators**

The *example obfuscating transformations* we have seen are instances of the 1st Futamura projection.

2nd Futamura projection: if $\mathtt{P}$ **is** $\mathtt{interp}^{\mathtt{flat}}$**, then** `compiler` **is a *stand-alone obfuscator*: a "flattening" program transformer.**

We have done all three using the UNMIX **specializer.**

IMMEDIATE CONSEQUENCES OF FUTAMURA PROJ'S

There are other better ways to obfuscate and to produce a obfuscator:

- $\mathtt{p}' = [\![\mathtt{comp}]\!](\mathtt{p})$ **(obfuscate by compiler)** and
- $\mathtt{comp} = [\![\mathtt{cogen}]\!](\mathtt{interp}^+)$ **(generate obfuscator).**

Future developments will involve gaining a deeper understanding in expected time factors: the relations among

1. $time_{\mathtt{p}'}(\mathtt{d})$ **and** $time_{\mathtt{p}}(\mathtt{d})$**: the slowdown imposed by the obfuscation;**
2. $time_{\mathtt{spec}}(\mathtt{interp}^+, \mathtt{p})$ **and** $length(\mathtt{p})$**: the amount of time to do the obfuscation by general specialization;**
3. $time_{\mathtt{comp}}(\mathtt{p})$ **and** $length(\mathtt{p})$**: the amount of time to do the obfuscation by running a generated obfuscator (significantly less than above).**

ANOTHER TYPE OF OBFUSCATION

A conflict that makes program obfuscation a subtle problem concerns a general principle in programming:

Good programs are well-structured and have concise invariants.

This led to flattening, to:

obfuscation by making the program's control flow hard to understand.

There is another direction to attack the problem:

obfuscation by making the program's invariants and data flow hard to understand.

A next step: insertion of opaque predicates . Trick: replace command `C` **by**

```
if always-false-test then junk-commands else C
```

Point: the inserted test and `then`**-branch code will (barely) affect the program execution. But they will complicate life for a program attacker that does not know the program's semantics.**

SUMMING UP

1. Nice (from my viewpoint): program obfuscation provides a motive to do automatic program transformation
2. Different criteria for success from traditional program transformation:
   - not for increased efficiency,
   - not to compile, i.e., to change the programming language;
   - but to make programs hard to understand or adapt by other people (or by their automated program attack systems)
3. Partial evaluation provides a well-developed approach to do automatic program transformation
4. Starting point: a "vanilla" self-interpreter for the source language
5. This is then distorted, so its specialization to a clear-text source program will produce a computationally equivalent and correct but harder-to-understand target program.
6. Examples: data distortion; flattening; inserting opaque predicates.

SOME REFERENCES

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, pp. 144–155. Springer, 2001.

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

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

S. Drape. Obfuscation of Abstract Data-Types. PhD thesis, University of Oxford, 2004.

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

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

C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. IEEE Int. Conf. Dependable Systems and Networks, pp. 193–202, 2001.

QUESTIONS ?