New approaches for chasing metamorphic malware

Isabella Mastroeni
University of Verona, Italy
Joint work with Roberto Giacobazzi, Neil Jones, Mila Dalla Preda

30 May 2013

ESCAPE SIGNATURE CHECKING

Polymorphic malware

The malware code is encrypted and contains a decryption routine that decrypts the code and then executes it.

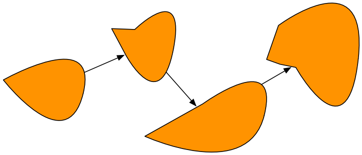
ESCAPE SIGNATURE CHECKING

Polymorphic malware

The malware code is encrypted and contains a decryption routine that decrypts the code and then executes it.

Metamorphic malware

The malware applies semantics-preserving transformations (e.g. obfuscations) to mutate its own code as it propagates.



ATTACKING METAMORPHISM

Our research directions

Metamorphism is mainly based on obfuscation techniques:

- We can study obfuscation techniques
- We can extract behavioural malware characterizations

ATTACKING METAMORPHISM

Our research directions

Metamorphism is mainly based on obfuscation techniques:

- We can study obfuscation techniques
  - Different from reverse engineering: we are not interested in the original code, we look for properties characterizing semantic invariants;
- We can extract behavioural malware characterizations

ATTACKING METAMORPHISM

Our research directions

Metamorphism is mainly based on obfuscation techniques:

- We can study obfuscation techniques
  - Different from reverse engineering: we are not interested in the original code, we look for properties characterizing semantic invariants;
- We can extract behavioural malware characterizations
  - We can use higher-order (abstract) non-interference properties for characterizing the interaction of malware with the environment;
  - Further application: We can study how to defeat anti-emulation techniques.

EXAMPLE

(Pseudo-)Code:

```
mov eax, [edx+0Ch]
push ebx
push [eax]
call ReleaseLock
```

EXAMPLE

(Pseudo-)Code:

```
mov eax, [edx+0Ch]
push ebx
push [eax]
call ReleaseLock
```

Obfuscated code (junk):

```
mov eax, [edx+0Ch]
inc eax
push ebx
dec eax
push [eax]
call ReleaseLock
```

Example

(Pseudo-)Code:

```
mov eax, [edx+0Ch]
push ebx
push [eax]
call ReleaseLock
```

Obfuscated code (junk + reordering):

```
mov eax, [edx+0Ch]
jmp +3
push ebx
dec eax
jmp +4
inc eax
jmp -3
call ReleaseLock
jmp +2
push [eax]
jmp -2
```

PROTECTION BY OBSCURITY

$\mathfrak{O} : \mathbb{P} \rightarrow \mathbb{P}$ is a code obfuscator if it is an obfuscating compiler:

- It is potent: $\mathfrak{O}(P)$ is *more complex* (ideally unintelligible) than P;
- It preserves the observational behaviour of programs $[\![\mathfrak{O}(P)]\!] = [\![P]\!]$ [C. Collberg et al. '97, '98]

The limit. Obfuscating programs is (im)possible:

Even under restrictive hypothesis a general purpose obfuscator generating perfectly unintelligible code (virtual black-box) does not exist! [Barak et al. '01]

The challenge. Design obfuscators that work against specific attacks

Extensional properties of programs are undecidable [Rice '53]
....so formal methods and static analysis are born!

APPROXIMATION VS OBSCURITY

- Because of undecidability we need approximation
- Even if decidable, it is typically too complex to trace/analyze/understand (500kC $\sim$ 600 mY) so we need approximation
- Approximation is pervasive in computing and code understanding

There are only approximated interpretations of programs

- Making obscure is making the approximated interpreter blind!
- Potent obscure transformations correspond to hardly improvable approximations

How can we formalize all this?

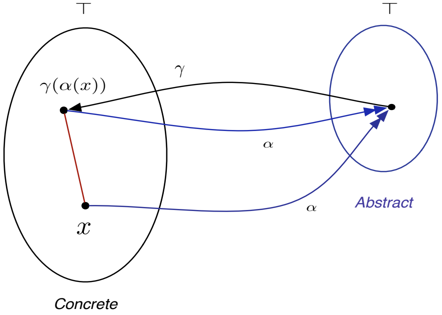
WHY ABSTRACT INTERPRETATION?

Abstract Interpretation (1977) is the a general model for the (static or dynamic) approximation of semantics of discrete dynamic systems

- Including: Static program analysis, dynamic analysis, profiling, debugging, tracing, compilation, de-compilation, type checking and type inference, model checking and predicate abstraction, trajectory evaluation, testing, proof systems, etc.

ABSTRACT INTERPRETATION

Design approximate semantics of programs [Cousot & Cousot '77, '79].



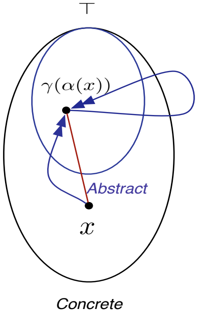
Galois Connection: $\langle C, \alpha, \gamma, A \rangle$, A and C are complete lattices.

Closures: $\langle uco(C), \sqsubseteq \rangle$ set of all possible abstract domains,

$A_1 \sqsubseteq A_2$ if A_1 is more concrete than A_2

ABSTRACT INTERPRETATION

Design approximate semantics of programs [Cousot & Cousot '77, '79].



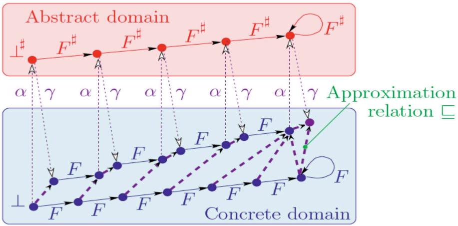
$\gamma \circ \alpha \in uco(C)$

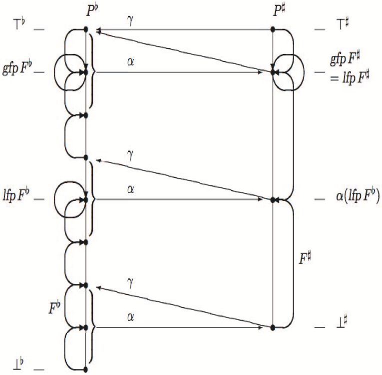
Galois Connection: $\langle C, \alpha, \gamma, A \rangle$, A and C are complete lattices.

Closures: $\langle uco(C), \sqsubseteq \rangle$ set of all possible abstract domains,

$A_1 \sqsubseteq A_2$ if A_1 is more concrete than A_2

APPROXIMATING INTERPRETATION: BCA





G is a sound approximation of F if

$$\alpha \circ F \circ \gamma \sqsubseteq G$$

SOUNDNESS AND COMPLETENESS

[Cousot & Cousot '79]

- A program $P \in \mathbb{P}$ and a domain of computation C
- An interpreter: $[\![\cdot]\!] : \mathbb{P} \times C \longrightarrow C$
- (Approximate) observable properties: $\rho = \gamma \circ \alpha \in uco(C)$
- DERIVE A SOUND APPROXIMATE SPECIFICATION $[\![P]\!]^\sharp$

$$\rho([\![P]\!](x)) \leq [\![P]\!]^\sharp(x)$$

- THE LIMIT CASE: COMPLETENESS

$$\rho([\![P]\!](x)) = [\![P]\!]^\sharp(x) \text{ iff } \rho([\![P]\!](x)) = \rho([\![P]\!](\rho(x)))$$

SOUNDNESS AND COMPLETENESS

- `WhichChess` : $Img \longrightarrow \wp(Chess)$ returns the type of chess on the chessboard.
- $\rho : Img \longrightarrow Img$ such that: ρ () = 
- $\eta : \wp(Chess) \longrightarrow [0, 12]$ counts the number of different types of chess

η (`WhichChess` (ρ ())) = η (`WhichChess` ())

$= 12$

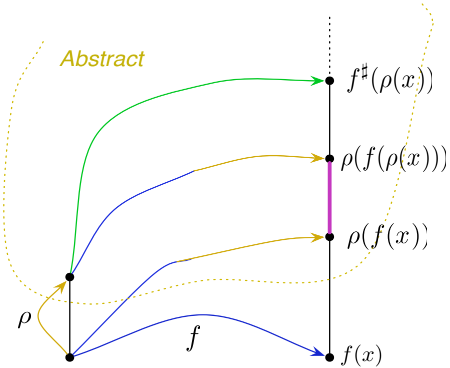
$\geq$ η (`WhichChess` ())

$= 7$

COMPLETENESS IN ABSTRACT INTERPRETATION

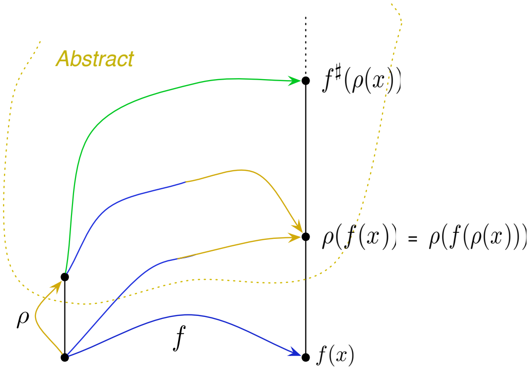
- BACKWARD SOUNDNESS:
  NO INFORMATION IS LOST BY APPROXIMATING THE INPUT/OUTPUT

- $\rho \circ f \leq \rho \circ f \circ \rho$



COMPLETENESS IN ABSTRACT INTERPRETATION

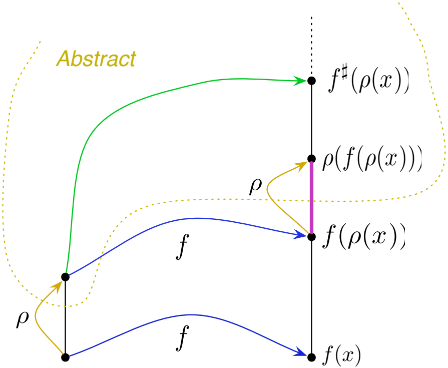
- BACKWARD COMPLETENESS:
  NO LOSS OF PRECISION IS ACCUMULATED BY APPROXIMATING THE INPUT
- $\rho \circ f = \rho \circ f \circ \rho$



COMPLETENESS IN ABSTRACT INTERPRETATION

- FORWARD COMPLETENESS:
  NO INFORMATION IS LOST BY APPROXIMATING THE OUTPUT

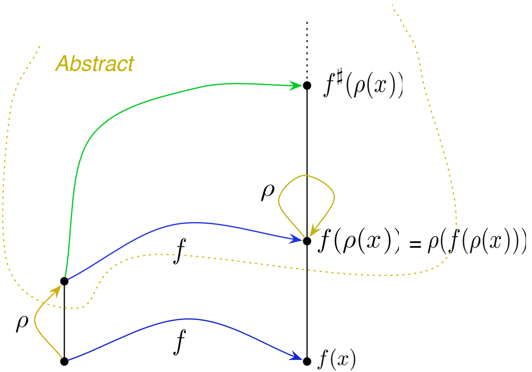
- $f \circ \rho \leq \rho \circ f \circ \rho$



COMPLETENESS IN ABSTRACT INTERPRETATION

- FORWARD COMPLETENESS:
  NO INFORMATION IS LOST BY APPROXIMATING THE OUTPUT

- $f \circ \rho = \rho \circ f \circ \rho$



OBSCURITY AS INCOMPLETENESS

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

- Let $\rho \in uco(\Sigma)$ with Σ semantic objects (data, traces etc)
- A program transformation $\tau : \mathbb{P} \to \mathbb{P}$ such that $[\![P]\!] = [\![\tau(P)]\!]$.
- ρ $\mathcal{B}$-complete for $[\![\cdot]\!]$ if $\rho([\![P]\!]) = [\![P]\!]^\rho$

τ obfuscates P if $[\![P]\!]^\rho \sqsubset [\![\tau(P)]\!]^\rho$

$$[\![P]\!]^\rho \sqsubset [\![\tau(P)]\!]^\rho \iff \rho([\![\tau(P)]\!]) \sqsubset [\![\tau(P)]\!]^\rho$$

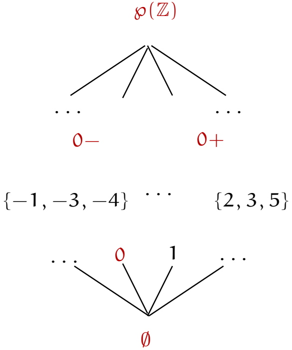
OBSCURITY AS INCOMPLETENESS

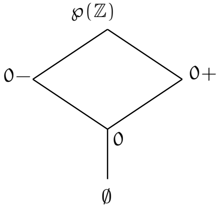
Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

$P : \mathtt{x} = \mathtt{a} * \mathtt{b}$

$Sign$ is an obvious abstraction of $\wp(\mathbb{Z})$:





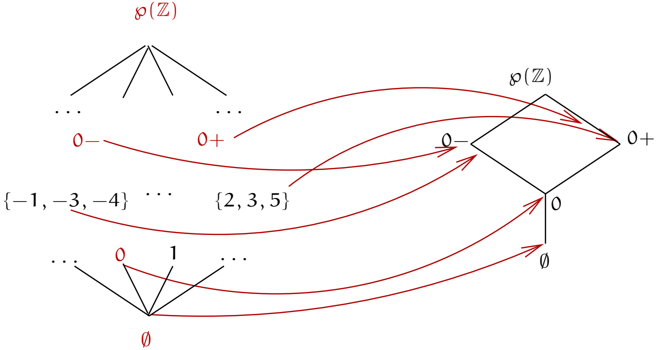
OBSCURITY AS INCOMPLETENESS

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

$P : \mathtt{x} = \mathtt{a} * \mathtt{b}$

$Sign$ is an abstraction of $\wp(\mathbb{Z})$:



OBSCURITY AS INCOMPLETENESS

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

$$P: \quad \mathtt{x = a * b} \quad \longrightarrow \quad \tau(P): \quad \begin{array}{l} \mathtt{x} = 0; \\ \mathtt{if}\ \mathtt{b} \leq 0\ \mathtt{then}\ \{\mathtt{a} = -\mathtt{a}; \mathtt{b} = -\mathtt{b}\}; \\ \mathtt{while}\ \mathtt{b} \neq 0\ \{\mathtt{x} = \mathtt{a} + \mathtt{x}; \mathtt{b} = \mathtt{b} - 1\} \end{array}$$

- $Sign$ is complete for P:
  - ✔ $[\![P]\!]^{Sign} = \lambda \mathsf{a}, \mathsf{b}.\ Sign(\mathsf{a} * \mathsf{b})$
- $Sign$ is incomplete for $\tau(P)$:
  - ✔ $[\![\tau(P)]\!]^{Sign} = \lambda \mathsf{a}, \mathsf{b}. \begin{cases} 0 & \text{if } \mathsf{a} = 0 \vee \mathsf{b} = 0 \\ \wp(\mathbb{Z}) & \text{otherwise} \end{cases}$
- Is there any way to get $\tau(P)$ systematically out of P?

EXPLOITING INCOMPLETENESS

Maximize $[\![P]\!]^\rho$ incompleteness!

- The abstraction is the specification of the attacker
  - ✔ Profiling: Abstract memory keeping only (partial) resource usage
  - ✔ Tracing: Abstraction of traces (e.g., by trace compression)
  - ✔ Slicing: Abstraction of traces (relative to variables)
  - ✔ Monitoring: Abstraction of trace semantics ([Cousot&Cousot POPL02])
  - ✔ Decompilation: Abstracts syntactic structures (e.g., reducible loops)
  - ✔ Disassembly: Abstracts binary structures (e.g., recursive traversal)
- Each abstraction is incomplete for a concrete enough trace semantics
- Maximize incompleteness by code transformation: Obfuscation
- Exploit incompleteness for hiding information: Steganography

THE IDEA [GIACOBAZZI, JONES & MASTROENI '12]

Build a *general-purpose program transformer* by programming a self-interpreter in a style to give the desired transformation

CLAIM: $[\![\mathtt{P}]\!] = [\![\mathtt{P}']\!]$, by simple equational reasoning:

$[\![\mathtt{P}]\!](\mathtt{d})$	$=$	$[\![\mathtt{interp}]\!](\mathtt{P},\mathtt{d})$	definition of self-interpreter
	$=$	$[\![[\![\mathtt{spec}]\!](\mathtt{interp},\mathtt{P})]\!](\mathtt{d})$	definition of specializer
	$=$	$[\![\mathtt{P'}]\!](\mathtt{d})$	definition of P'

Therefore the function

$$\mathtt{P} \longmapsto [\![\mathtt{spec}]\!](\mathtt{interp},\mathtt{P})$$

is a *semantics-preserving program transformer*!!

We need to *change the interpretation*: $\mathtt{interp} \rightsquigarrow \mathtt{interp}^{+}$

An easy example: Data obfuscation

Similar to Drape 2004 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 $dob(x)$ obfuscate or invert obfuscation.
- We consistently modify `interp` so that:
  - ✔ input values are obfuscated in the initial store;
  - ✔ variable values are obfuscated just before putting in the store;
  - ✔ output values are de-obfuscated in the program's final store;
  - ✔ expression evaluation yields non-obfuscated values:
    - » constant values are not obfuscated,
    - » variables' values must be de-obfuscated when got from the store

AN EASY EXAMPLE: THE INTERPRETER

input P, d; — Program to be interpreted, and *its* data

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

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

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

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

  case $instruction$ **of** — Dispatch on syntax

  skip : $pc := pc + 1$; — Obfuscate values when stored:

  $x := e$: $store := store[x \mapsto obf(eval(e, store))]$; $pc := pc + 1$;

  $\ldots$ **endw** ;

output $dob(store[out])$;

$obf(V) = 2 * V$; $dob(V) = V/2$ — Obfuscation/de-obfuscation

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

  $constant$: $obf(e)$

  $variable$: $dob(store(e))$ — De-obfuscate variable values

  $e1 + e2$: $eval(e1, store) + eval(e2, store)$

  $e1 - e2$: $eval(e1, store) - eval(e2, store)$

  $\ldots$

AN EASY EXAMPLE: THE OUTPUT

The source program is automatically transformed into this equivalent obfuscated one

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

SIGN ANALYSIS

- Sign analysis is complete for multiplication $*$: exact information.
- Sign analysis is incomplete for addition $+$: imprecise information

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

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

Our trick: ...let the interpreter evaluate!

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

SIGN ANALYSIS

- Sign analysis is complete for multiplication $*$: exact information.
- Sign analysis is incomplete for addition $+$: imprecise information

P:

```
1. input x;
2. y := 2;
3. while x > 0 do
     4. y := y * y;
     5. x := x - 1
   endw
6. output y;
7. end
```

$\longmapsto$

P':

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

Sign analysis yields $y \mapsto +$ in P, but it yields $y \mapsto \top$ in P'.

The big goal

- A deep relation between obfuscation and interpretation
- Attack and defense are two aspects of interpretation
- Define a uniform framework for information concealment in programming languages
  - ✔ General enough to include most known methods
  - ✔ Formal enough to provide a (possibly) provable secure environment for obfuscation (and steganography) relatively to a fixed attacker
  - ✔ Rich enough to provide advanced design and evaluation methods
  - ✔ Practical enough to generate truly obfuscated
- The goal: *develop a theory and practice for code obfuscation (and steganography) in order to make these technologies as practical as analogous ones in other media (e.g., in DRM of audio and video)*

Completeness and metamorphism

Obfuscation is incompleteness

Obfuscation deceives all analyses incomplete wrt the made transformation

Hence...

Incompleteness transformers characterise the set of deceived analyses! [Giacobazzi & Mastroeni '12]

Metamorphism is obfuscation

Malware protects its code by using obfuscation techniques.

Hence...

Completeness transformers characterises the set of successful malware detection analyses?

Malware detection

Malware detector

$$\mathcal{D}(P, M) = \begin{cases} \textit{true} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \textit{false} & \text{otherwise} \end{cases}$$

MALWARE DETECTION

Malware detector

$$\mathcal{D}(P, M) = \begin{cases} \textit{true} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \textit{false} & \text{otherwise} \end{cases}$$

An ideal malware detector is sound and complete:

- SOUND = no false positives (no false alarms)

MALWARE DETECTION

Malware detector

$$\mathcal{D}(P, M) = \begin{cases} \textit{true} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \textit{false} & \text{otherwise} \end{cases}$$

An ideal malware detector is sound and complete:

- SOUND = no false positives (no false alarms)
- COMPLETE = no false negatives (no missed alarms)

CHASING METAMORPHISM

In order to detect metamorphic malware variants malware detector should be based on SEMANTIC program features.

CHASING METAMORPHISM

In order to detect metamorphic malware variants malware detector should be based on SEMANTIC program features.

[Dalla Preda et al ’07]

Formal framework for malware detection based on program semantics and abstract interpretation.

CHASING METAMORPHISM

In order to detect metamorphic malware variants malware detector should be based on SEMANTIC program features.

[Dalla Preda et al '07]

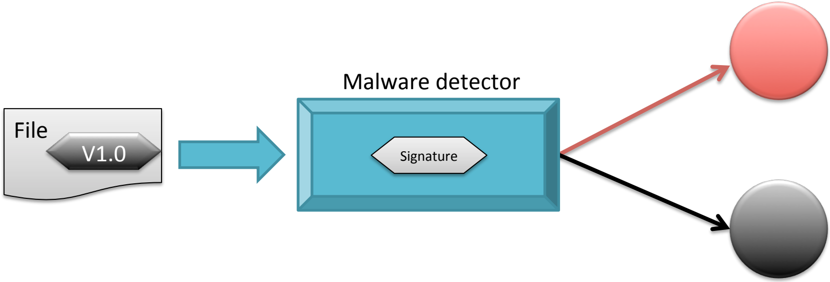
Formal framework for malware detection based on program semantics and abstract interpretation.

LIMIT

It assumes that the malware APPENDS its code and behaviour to the target program without interacting with it

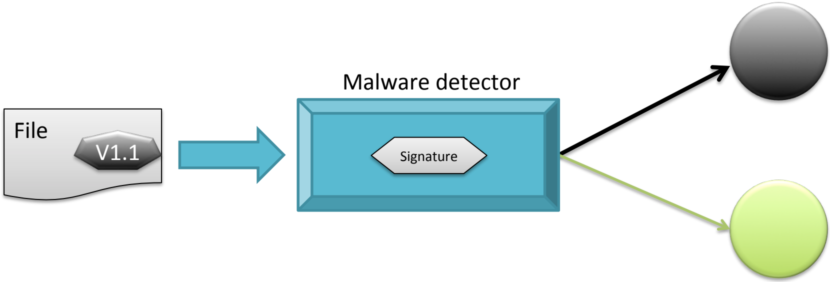
HOANI AND MD: THE IDEA

Metamorphism *defeats* the malware detector if it does generate an INTERFERENCE!



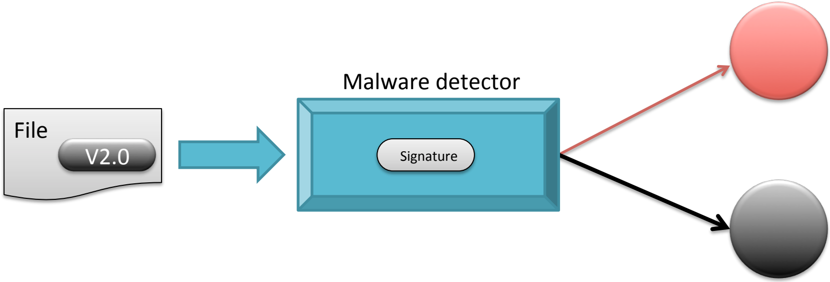
HOANI AND MD: THE IDEA

Metamorphism *defeats* the malware detector if it does generate an INTERFERENCE!



HOANI AND MD: THE IDEA

Metamorphism *defeats* the malware detector if it does generate an INTERFERENCE!



HOANI AND MD

IDEA

Define a more general framework for metamorphic malware infection where it is possible to express the interactions between different code fragments (e.g. the viral code and the target program)

HOANI AND MD

IDEA

Define a more general framework for metamorphic malware infection where it is possible to express the interactions between different code fragments (e.g. the viral code and the target program)

[Sabelfed and Mayers '03]

Non-interference (NI) reasons on data dependencies

HOANI AND MD

IDEA

Define a more general framework for metamorphic malware infection where it is possible to express the interactions between different code fragments (e.g. the viral code and the target program)

[Sabelfed and Mayers '03]

Non-interference (NI) reasons on data dependencies

[Giacobazzi and Mastroeni '04]

Abstract non-interference (ANI) generalizes NI by weakening the dependences between data

HOANI AND MD

IDEA

Define a more general framework for metamorphic malware infection where it is possible to express the interactions between different code fragments (e.g. the viral code and the target program)

[Sabelfed and Mayers '03]

Non-interference (NI) reasons on data dependencies

[Giacobazzi and Mastroeni '04]

Abstract non-interference (ANI) generalizes NI by weakening the dependences between data

High Order ANI (HOANI): Lift the ANI framework to programs.

MALWARE DETECTION

Malware detector

$$\mathcal{D}(P, M) = \begin{cases} \textit{true} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \textit{false} & \text{otherwise} \end{cases}$$

MALWARE DETECTION

Malware detector

$$\mathcal{D}(P, M) = \begin{cases} \textit{true} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \textit{false} & \text{otherwise} \end{cases}$$

- Consider a set $\mathbb{O}$ of obfuscating transformations ranged over by $\mathcal{O}$.
- Let $M \hookrightarrow P$ denote that program P is infected with malware M.

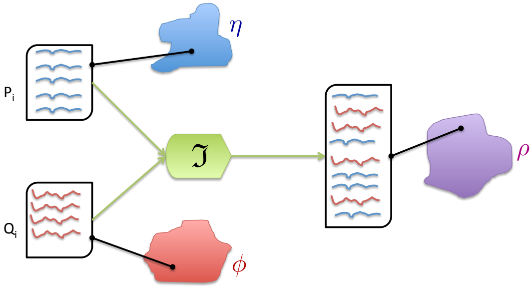
Relative soundness and completeness

$\mathcal{D}$ is SOUND for $\mathbb{O}$ if $\mathcal{D}(P, M) = \textit{true} \;\Rightarrow\; \exists \mathcal{O} \in \mathbb{O} : \mathcal{O}(M) \hookrightarrow P$

$\mathcal{D}$ is COMPLETE for $\mathbb{O}$ if $\forall \mathcal{O} \in \mathbb{O} : \mathcal{O}(M) \hookrightarrow P \;\Rightarrow\; \mathcal{D}(P, M) = \textit{true}$

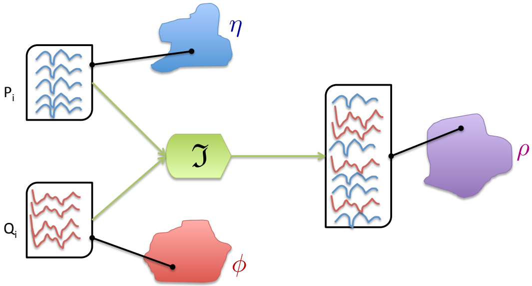
HOANI

$$[\![P_1]\!]^\eta = [\![P_2]\!]^\eta \wedge [\![Q_1]\!]^\phi = [\![Q_2]\!]^\phi \Rightarrow [\![\mathfrak{I}(Q_1, P_1)]\!]^\rho = [\![\mathfrak{I}(Q_2, P_2)]\!]^\rho$$



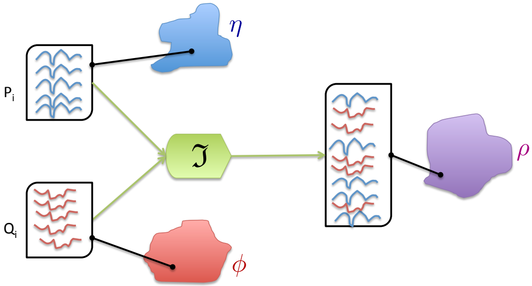
HOANI

$$[\![P_1]\!]^\eta = [\![P_2]\!]^\eta \wedge [\![Q_1]\!]^\phi = [\![Q_2]\!]^\phi \Rightarrow [\![\mathfrak{I}(Q_1, P_1)]\!]^\rho = [\![\mathfrak{I}(Q_2, P_2)]\!]^\rho$$



HOANI

$$[\![P_1]\!]^{\eta} = [\![P_2]\!]^{\eta} \wedge [\![Q_1]\!]^{\phi} = [\![Q_2]\!]^{\phi} \Rightarrow [\![\mathfrak{I}(Q_1, P_1)]\!]^{\rho} = [\![\mathfrak{I}(Q_2, P_2)]\!]^{\rho}$$



HOANI-BASED MD

- $P \in \mathit{Progr}$, $[\![P]\!]$ its (concrete) semantics on the domain C
- ρ property on Progr, $[\![P]\!]^\rho$ the abstract semantics of program P

HOANI-BASED MD

- $P \in \mathit{Progr}$, $[\![P]\!]$ its (concrete) semantics on the domain C
- ρ property on Progr, $[\![P]\!]^\rho$ the abstract semantics of program P

ANIMD

$$\mathit{ANIMD}_\rho(M,P) = \mathit{true} \Leftrightarrow \exists T \in \mathit{Progr} : [\![\mathfrak{I}(M,T)]\!]^\rho = [\![P]\!]^\rho$$

HOANI-BASED MD

- $P \in Progr$, $[\![P]\!]$ its (concrete) semantics on the domain C
- ρ property on $Progr$, $[\![P]\!]^\rho$ the abstract semantics of program P

ANIMD

$ANIMD_\rho(M,P) = true \Leftrightarrow \exists T \in Progr : [\![\mathfrak{I}(M,T)]\!]^\rho = [\![P]\!]^\rho$

Metamorphic engine (ME)

Let ϕ the semantic property preserved by the ME:

$$\mathbb{O}_\phi = \left\{ \mathcal{O} \,\middle|\, \forall M, M_1 \in Prog : [\![M]\!]^\phi = [\![M_1]\!]^\phi \Leftrightarrow M_1 = \mathcal{O}(M) \right\}$$

HOANI-BASED MD

- $P \in Progr$, $[\![P]\!]$ its (concrete) semantics on the domain C
- ρ property on $Progr$, $[\![P]\!]^\rho$ the abstract semantics of program P

ANIMD

$$ANIMD_\rho(M,P) = true \Leftrightarrow \exists T \in Progr : [\![\Im(M,T)]\!]^\rho = [\![P]\!]^\rho$$

Metamorphic engine (ME)

Let ϕ the semantic property preserved by the ME:

$$\mathbb{O}_\phi = \left\{ \mathcal{O} \;\middle|\; \forall M, M_1 \in Prog : [\![M]\!]^\phi = [\![M_1]\!]^\phi \Leftrightarrow M_1 = \mathcal{O}(M) \right\}$$

$HOANI_\rho^\phi$

$$[\![M]\!]^\phi = [\![M_1]\!]^\phi \Rightarrow [\![\Im(M,T)]\!]^\rho = [\![\Im(M_1,T)]\!]^\rho$$

WHAT CAN WE DO?

CERTIFYING *MD*

We can characterize the **most concrete** property ϕ such that *ANIMD* is SOUND and COMPLETE for $\mathbb{O}_\phi$!

What can we do?

Certifying *MD*

We can characterize the **most concrete** property ϕ such that *ANIMD* is Sound and Complete for $\mathbb{O}_\phi$!

Training *MD*

Given $\mathbb{O}_\phi$ we can characterize the **most concrete** property ρ such that *ANIMD*$_\rho$ is Complete for $\mathbb{O}_\phi$!

WHAT CAN WE DO?

CERTIFYING *MD*

We can characterize the **most concrete** property ϕ such that *ANIMD* is SOUND and COMPLETE for $\mathbb{O}_\phi$!

TRAINING *MD*

Given $\mathbb{O}_\phi$ we can characterize the **most concrete** property ρ such that $ANIMD_\rho$ is COMPLETE for $\mathbb{O}_\phi$!

SMD_ρ [Dalla Preda et al. '07]

$$SMD_\rho(M, P) = true \quad \Leftrightarrow$$
$$\exists Q, T \in Progr : [\![P]\!] = [\![\mathfrak{I}(Q, T)]\!] \wedge \rho([\![M]\!]) = \rho([\![Q]\!])$$

WHAT CAN WE DO?

CERTIFYING *MD*

We can characterize the **most concrete** property ϕ such that *ANIMD* is SOUND and COMPLETE for $\mathbb{O}_\phi$!

TRAINING *MD*

Given $\mathbb{O}_\phi$ we can characterize the **most concrete** property ρ such that $ANIMD_\rho$ is COMPLETE for $\mathbb{O}_\phi$!

SMD_ρ [Dalla Preda et al. '07]

$$SMD_\rho(M, P) = true \;\; \Leftrightarrow$$
$$\exists Q, T \in Progr : [\![P]\!] = [\![\mathfrak{I}(Q, T)]\!] \wedge \rho([\![M]\!]) = \rho([\![Q]\!])$$

WHAT'S NEW IN *ANIMD*

$ANIMD_\rho(M, P)$ is more general than $SMD_\rho(M, P)$.

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI
  - Understand and develop *HOANI* and its application to MD;

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI
  - Understand and develop *HOANI* and its application to MD;
  - Develop a systematic *strategy* for the design of the best MD given a class of code variants

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI
  - Understand and develop *HOANI* and its application to MD;
  - Develop a systematic *strategy* for the design of the best MD given a class of code variants
    - Develop a technique for learning the ME that generates a given set of variants;

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI
  - Understand and develop *HOANI* and its application to MD;
  - Develop a systematic *strategy* for the design of the best MD given a class of code variants
    - Develop a technique for learning the ME that generates a given set of variants;
    - Understand how to generate the invariant property ϕ of ME;

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI
  - Understand and develop *HOANI* and its application to MD;
  - Develop a systematic *strategy* for the design of the best MD given a class of code variants
    - Develop a technique for learning the ME that generates a given set of variants;
    - Understand how to generate the invariant property ϕ of ME;
    - Derive the observation property ρ that characterizes detection for ANIMD_ρ;

FUTURE WORKS

- Obfuscation and metamorphism
  - Understand how completeness can help in defeating metamorphism;
- Malware and HOANI
  - Understand and develop *HOANI* and its application to MD;
  - Develop a systematic *strategy* for the design of the best MD given a class of code variants
    - Develop a technique for learning the ME that generates a given set of variants;
    - Understand how to generate the invariant property ϕ of ME;
    - Derive the observation property ρ that characterizes detection for $ANIMD_\rho$;
- This approach can be used for avoiding anti-emulation techniques used by modern malware [Dinaburg et al. '08, Kang et al. '09].