Modeling metamorphosis by abstract interpretation

Roberto Giacobazzi

16.09.2010 - SAS2010
The problem
Malware analysis: signature checking

- **Malware** refers to malicious software
- **Signature checking**: identify a sequence of instructions which is unique to a malware (virus signature) then scan program for signatures
- **Example**: *Chernobyl* signature:
  
  ```
  E800 0000 005B 8D4B 4251 5050
  OF01 4C24 FE5B 83C3 1CFA 882B
  ```
- Cumbersome, inaccurate, easy to foil....
Anti-anti malware

- How can we escape signature checking?
  - *by dynamically modifying malware structure!*
- **Polymorphic malware** contain decryption routines which decrypt encrypted constant parts of their body.
- **Metamorphic malware** typically do not use encryption, but mutates (obfuscate) forms in subsequent generations.
Metamorphism as obfuscation

**From Chernobyl CIH 1.4**

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop ecx</td>
<td>pop ecx</td>
</tr>
<tr>
<td>jecxz SFModMark</td>
<td>jecxz SFModMark</td>
</tr>
<tr>
<td>mov esi, ecx</td>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td>mov eax, 0d601h</td>
<td>beqz N1</td>
</tr>
<tr>
<td>pop edx</td>
<td>mov esi, ecx</td>
</tr>
<tr>
<td>pop ecx</td>
<td>nop</td>
</tr>
<tr>
<td>call edi</td>
<td>mov eax, 0d601h</td>
</tr>
<tr>
<td>jmp Loop</td>
<td>pop edx</td>
</tr>
<tr>
<td></td>
<td>pop ecx</td>
</tr>
<tr>
<td></td>
<td>jecxz SFModMark</td>
</tr>
<tr>
<td></td>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td></td>
<td>beqz N2</td>
</tr>
<tr>
<td></td>
<td>mov esi, ecx</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>mov eax, 0d601h</td>
</tr>
<tr>
<td></td>
<td>pop edx</td>
</tr>
<tr>
<td></td>
<td>pop ecx</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>call edi</td>
</tr>
<tr>
<td></td>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td></td>
<td>beqz N2</td>
</tr>
<tr>
<td></td>
<td>jmp Loop</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Chernobyl CIH 1.4

Thursday, September 16, 2010
Metamorphism as obfuscation

From Chernobyl CIH 1.4

Loop:

pop ecx
nop
call edi
xor ebx, ebx
beqz N2
N2:
jmp Loop

nop
mov eax, 0d601h
pop edx
pop ecx
call edi
jmp Loop

jecxz SFModMark
mov esi, ecx

N1:
mov esi, ecx
Metamorphism as obfuscation

From Chernobyl CIH 1.4

Loop:

pop ecx
jecxz SFModMark
mov esi, ecx
mov eax, 0d601h
pop edx
pop ecx
call edi
jmp Loop

L1:
jecxz SFModMark
xor ebx, ebx
beqz N1
mov esi, ecx
jmp L2

N1:
mov esi, ecx
jmp L2

L2:
nop
mov eax, 0d601h
pop edx
pop ecx
nop
jmp L3

L3:
call edi
xor ebx, ebx
beqz N2
jmp L4

N2:
jmp Loop

L4:
From Chernobyl CIH 1.4

Thursday, September 16, 2010
Metamorphism: an example

Malware evolution

- How can we model and compute signatures for metamorphosis?
Metamorphism: some (public) history

http://vx.netlux.org/

Win32.Evol
swaps instructions with equivalents
inserts junk code between essential instructions

Regswap (Win32)
same code different register names

BadBoy (DOS) and Ghost (Win32)
same code different subroutine order (n! possible mutations: 10 modules ~3.6M possible signatures)

Zmorph (Win95)
decrypt virus body instruction by instruction
push instructions on stack
insert and remove jumps
rebuild body on stack

by Peter Szor

Zperm (Win95)

---------------------------------------------
Attacking metamorphism

* **Idea:** Behavior Monitors

* Run suspect program in an emulator and extract a DB of relevant signatures (huge DB)

* Look for changes in file structure: Some viruses modify files in a consistent way (inaccurate)

* Disassemble and look for virus-like instructions: reverse engineering malware (expensive)
Problems

• The code may contain its own metamorphic engine ME

• The metamorphic engine can be used when engineering malware

• Metamorphic signature: is a language $\mathcal{L}$ of possible signatures generated by a metamorphic malware:

\[ \sigma \in \mathcal{L} \Rightarrow \sigma \text{ is a possible signature} \]

• Is there a way for extracting a metamorphic signatures?
Related works

- Specify some abstraction (CFG, instruction equivalence, rewrite rules towards normal form - undo metamorphism)

- [Dalla Preda et al POPL07, Filiol PWASET07, Zbitsky JCV 09, Bonfante et al JCV 09]

- Existing semantics-based approach to malware detection are promising but they still rely on a priori knowledge of the metamorphic transformations used by malware writers

- Need to model the self-modifying behavior of a metamorphic malware without any a priori knowledge of the transformations it uses
Idea: Extract $L$ as a abstract interpretation of the metamorphic malware!

Extracting metamorphic signatures is approximating malware semantics

- data objects are code slices
- abstraction acts on code structure (code may be as complex as data!!)
- invariants on mutational code structure describe the metamorphic engine behavior!!
- fix-point abstraction approximate invariants, i.e. generates metamorphic signatures....
Modeling metamorphism
Phase semantics

- States: no distinction between code and data

- Phase semantics: partition the trace of execution states into phases, each collecting the computation of a particular code variant

- Maximal trace semantics: \( S[P] = \text{lfp}_T \mathcal{F}[P] \)

\[
\text{bound}(s) = \{s_0\} \cup \{s_i \mid \text{MOD}(s_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}
\]

\[
\text{phases}(s) = \{s_i \ldots s_j \mid s_i, s_{j+1} \in \text{bound}(s), \forall l \in [i+1, j] : s_l \not\in \text{bound}(s)\}\]
Fix-point phase semantics

- Program evolution graph: $G[P_0] = (V, E)$
  - Nodes = Phases
  - Edges = Phase transitions
- The phase semantics of a program $P_0$ is given by the set of all possible paths of its program evolution graph

$$S^{Ph}[P_0] = \{ P_0 \ldots P_n | \forall i \in [0, n - 1] : (P_i, P_{i+1}) \in E \}$$
Fix-point phase semantics

* Phase transition: \( T^{Ph}(P_0) = \left\{ P_i \mid s = s_0 \ldots s_i \ldots s_n \in S[P_0], s_i \in bound(s), \forall l \in [1, i - 1]: s_l \not\in bound(s) \right\} \)

* Fix-point iteration: \( S^{Ph}[P] = \operatorname{lfp} \mathcal{F}_{T^{Ph}}[P] \)
Correctness of phase semantics

\[ \langle \varphi(\Sigma^*), \subseteq \rangle \leftrightarrow_{\alpha_{Ph}} \langle \varphi(P^*), \subseteq \rangle \]

- Trace semantics and phase semantics are related by abstraction:
  - \( \alpha_{Ph} \) keeps only phase bounds

- Locally incomplete.....

- Fix-point complete:
  \[ \alpha_{Ph}(\text{lfp}\mathcal{F}_\mathcal{T}[P_0]) = \text{lfp}\mathcal{F}_\mathcal{T}^{Ph}[P_0] \]

CONCRETE TEST FOR METAMORPHISM

\[ P_0 \leadsto_{Ph} Q \iff \exists P_0, P_1, \ldots, P_n \in S^{Ph}[P_0], \exists i \in [0, n]: P_i = Q \]

no false positives, no false negatives
Abstracting metamorphism
Abstracting phases

* Need abstraction for approximating phases!!!

design GC: $\langle \wp(\wp^*) \rangle \equiv \left\langle \alpha_A, \subseteq \right\rangle$

define the abstract transition relation $T^A : A \rightarrow \wp(A)$

define $\mathcal{F}_{T^A}[P_0] : A \rightarrow A$ whose fixpoint computation

\[ \text{lfp}^A \mathcal{F}_{T^A}[P_0] = S^A[P_0] \]

corresponds to the abstract specification of the metamorphic behavior

prove that $S^A[P_0]$ is a correct approximation of phase semantics $S^{Ph}[P_0]$, i.e.,

\[ \alpha_A \left( \text{lfp}^A \mathcal{F}_{T^{Ph}}[P_0] \right) \subseteq_A \text{lfp}^A \mathcal{F}_{T^A}[P_0] \]

ABSTRACT TEST FOR METAMORPHISM

$P_0 \sim_A Q \iff \alpha_A(Q) \subseteq_A S^A[P_0]$ 

no false negatives
Phases as FSA

\[ \alpha : P \rightarrow \mathcal{F} \]

<table>
<thead>
<tr>
<th>P_0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>MEM[f] := 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8: MEM[MEM[f]] := MEM[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>if (MEM[a] mod 2) goto 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10: MEM[MEM[f] + 2] := encode(goto 6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>MEM[a] := (MEM[a] + 1)/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14: goto 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \hat{\alpha}(P_0) \]
Phase semantics as traces of FSA

\[ \alpha_{\tilde{\mathcal{S}}} \left( \text{ltf}_{\mathcal{F}T} \lceil P_0 \rceil \right) \subseteq \text{ltf}_{\mathcal{F}T} \tilde{\mathcal{S}} \lceil P_0 \rceil = S_{\tilde{\mathcal{S}}} \lceil P_0 \rceil \]
Phase semantics as traces of FSA: $S^\#[P_0]$

- We need a static approximation of the Phase transfer function
  - Stack analysis: approximating the values on top of the stack
  - Memory analysis: approximating the values stored in memory
- We emulate the run of a phase generating a superset of FSA that may be generated (over approximation!)

$$S^\delta[P_0] \subseteq S^\#[P_0]$$
Widening phases: regular metamorphism

- **Regular metamorphism**: mutation constrained in a regular language of instructions

- Collapsing a (static) trace of FSA into a single FSA: **widening**

- $\langle \mathcal{F}/\equiv, \sqsubseteq \mathcal{F} \rangle$ where $\mathcal{M}_1 \sqsubseteq \mathcal{F} \mathcal{M}_2 \iff L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$

\[
\begin{align*}
W_0 &= \hat{\alpha}(P_0) \\
W_{i+1} &= W_i \triangledown F^\cup_T [P_0](W_i)
\end{align*}
\]

**ABSTRACT TEST FOR METAMORPHISM on $\mathcal{F}/\equiv$**

$P_0 \sim_{\mathcal{F}} Q \iff \hat{\alpha}(Q) \sqsubseteq_{\mathcal{F}} W[P_0]$ no false negatives
Widening phases: regular metamorphism

* Let $M_1$ and $M_2$ be two FSA

* $R_n \subseteq Q_1 \times Q_2$ is a state relation

$(q_1, q_2) \in R_n$ if $q_1$ and $q_2$ recognize the same language of strings of length $n$

\[ q \equiv_R q' \iff \exists r \in Q_1 : (r, q) \in R_n \text{ and } (r, q') \in R_n \]

\[ M_1 \nabla M_2 = M_2 / \equiv_R \]

* It is a widening if on finite alphabet: approximate instruction terms!
Widening phases: regular metamorphosis

In [13] the authors use trace semantics to characterize the behaviours of both the malware and the potentially infected program, and use abstract interpretation to "hide" their irrelevant behaviours. A program is infected by a malware if their behaviours are indistinguishable up to a certain abstraction, which corresponds to obfuscations. A significant limitation of this work is that the knowledge of the obfuscation is essential in order to derive abstractions. In [19] the authors model the malware as a formula in the new logic CTPL, which is an extension of CTL able to handle register renaming. A program $P$ is infected by $M$, if $P$ satisfies the CTPL formula that models $M$. By

![Finite State Automaton](image)
Widening phases: regular metamorphosis

In [5], let us consider program $P_0$ obtained by enriching the metamorphic engine of program $P_0$ of Fig. 2 with a code permutation and data transformation that substitute instruction $\text{MEM}[e_1] := e_2$ with the equivalent sequence $\text{push } e_1, \text{pop } e_2$. A possible evolution is shown below, where $\text{ME}$ denotes the metamorphic engine.

$$
\begin{align*}
\text{MEM}[f] := 100; & \\
\text{input} \Rightarrow \text{MEM}[a]; & \\
\text{MEM}[a] \mod 2; & \\
\text{MEM}[a] := \text{MEM}[b]; \text{goto}; & \\
\text{MEM}[b] := \text{MEM}[a]; \text{goto}; & \\
\text{MEM}[a] := (\text{MEM}[a]+1)/2; \text{goto}; & \\
\text{MEM}[a] := \text{MEM}[a]/2; \text{goto}; & \\
\text{MEM}[a] := \text{MEM}[a]/2; \text{goto}; & \\
\text{MEM}[b] := \text{MEM}[a]; \text{goto}; & \\
\text{MEM}[a] := \text{MEM}[a]/2; \text{goto}; & \\
\text{MEM}[a] := \text{MEM}[a]/2; \text{goto}; & \\
\text{MEM}[a] := \text{MEM}[a]; \text{goto}; & \\
\text{MEM}[b] := \text{MEM}[a]; \text{goto}; & \\
\end{align*}
$$

Fig. 6 (b) shows the FSA that represents an approximation of all the possible evolutions of program $P_0$ when $k \geq 3$. This FSA is obtained through widening with widening seed $R_2$ and by applying the $\text{goto}$-reduction to handle permutation. We can observe that every time that in the automaton in Fig. 6 (b) we have an edge labeled with $\text{MEM}[e_1] := e_2$ between two states $q$ and $p$, there is also a path labeled with $\text{push } e_2, \text{pop } e_1$ that connects $q$ and $p$, and this precisely captures the fact that the metamorphic engine implements this substitution. The $\text{goto}$-reduction allows here to have a reduced FSA, and the self-loop labeled with $\text{nop}$ makes clear that the metamorphism could insert an unbounded number of $\text{nop}$ instructions.
Widening phases: regular metamorphosis

MEM[f]:=100; input=>MEM[a]; MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...

Fig. 6. Widened phase semantics.

MEM[f]:=100; input=>MEM[a]; MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...

Related Works and Discussion

In [13] the authors use trace semantics to characterize the behaviours of both the malware and the potentially infected program, and use abstract interpretation to "hide" their irrelevant behaviours. A program is infected by a malware if their behaviours are indistinguishable up to a certain abstraction, which corresponds to obfuscations. A significant limitation of this work is that the knowledge of the obfuscation is essential in order to derive abstractions. In [19] the authors model the malware as a formula in the new logic CTPL, which is an extension of CTL able to handle register renaming. A program P is infected by M if P satisfies the CTPL formula that models M. By
Widening phases: regular metamorphisn

MEM[f]:=100; input=>MEM[a]; MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...
Widening phases: regular metamorphosis

MEM[f]:=100; input=>MEM[a]; MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...

Fig. 6 (b) shows the FSA that represents an approximation of all the possible evolutions of program $P_0$ when $k \geq 3$. This FSA is obtained through widening with widening seed $R_2$ and by applying the $goto$-reduction to handle permutation. We can observe that every time that in the automaton in Fig. 6 (b) we have an edge labeled with $MEM[e_1]:=e_2$ between two states $q$ and $p$, then we have a path labeled with $push e_2, pop e_1$ that connects $q$ and $p$, and this precisely captures the fact that the metamorphic engine implements this substitution. The $goto$-reduction allows here to have a reduced FSA, and the self-loop labeled with $nop$ makes clear that the metamorphism could insert an unbounded number of $nop$ instructions.

Related Work and Discussion

In [13] the authors use trace semantics to characterize the behaviours of both the malware and the potentially infected program, and use abstract interpretation to "hide" their irrelevant behaviours. A program is infected by a malware if their behaviours are indistinguishable up to a certain abstraction, which corresponds to some obfuscations. A significant limitation of this work is that the knowledge of the obfuscation is essential in order to derive abstractions. In [19] the authors model the malware $M$ as a formula in the new logic CTPL, which is an extension of CTL able to handle register renaming. A program $P$ is infected by $M$, if $P$ satisfies the CTPL formula that models $M$. By $15$...
Example: code permutation + substitution

\[ P_0^+ : \]
\begin{align*}
1 & : \text{goto 8} \\
2 & : \text{if } (\text{MEM}[a] \mod 2) \text{ goto 11} \\
3 & : \text{nop} \\
4 & : \text{goto 100} \\
5 & : \text{push MEM}[a]/2 \\
6 & : \text{pop a} \\
7 & : \text{goto 12} \\
8 & : \text{MEM}[f] := 100 \\
9 & : \text{input } \Rightarrow \text{MEM}[a] \\
10 & : \text{goto 2} \\
11 & : \text{MEM}[a] := (\text{MEM}[a] + 1)/2 \\
12 & : \text{ME} \\
13 & : \text{goto 9} \\
100 & : \text{push MEM}[a] \\
101 & : \text{pop b} \\
102 & : \text{goto 5}
\end{align*}
Conclusions
What we have done!

What we have:

- A formal model of metamorphic code by Phase semantics
- A method for approximating the Phase semantics
- A computable approximation of regular metamorphism

The approach:

- requires **no a priori knowledge** about the metamorphic engine
- is **parametric** on several abstractions (instructions, phases, metamorphism...)
- is likely for **refinement** (grammars, constraints etc...)
- suitable for semi-automatic malware analysis: *generation-test-refine*
What is missing?

• An adequate experimental evaluation (beyond toy examples....)

• **Pro**: most malware implement relatively simple metamorphic engines (mostly regular) to foil syntactic signature checking

• **Con**: hacking can easily foil any abstraction

• A practical solution: behavioral monitoring + FSA abstraction + widening

• More advanced abstractions: e.g., *context free metamorphism & grammar widening*

• The paper is a preliminary approach to a **truly hard problem**!

• Next steps: experimental evaluation of regular metamorphism analysis, approximate behavioral monitoring.
Thanks!