Survey on Malware Detection on Binary

Hyunik Na

PLLab @ KAIST

ROSAEC 2nd Workshop

2009. 7. 9~11

Contents

- Introduction
  - Semantics approach to malware detection
  - Theoretical Limit in handling program semantics
- Two Existing Works Based on Semantics Signature
  - Template based
  - Model checking based
- Our Interest and Direction
- Q & A

Malware Detection

- What is malware?
  - Software containing malicious code
    - e.g. Virus, Worm, Trojan, Back door, Spyware
  - Spread through executable, script, or document, etc.

- Conventional malware detection
  - Syntactic (bit-pattern) signature matching
  - State of the art for most commercial detectors

```
E8 00000000          call 0h
5B                   pop ebx
8D 4B 42             lea ecx, [ebx+42h]
51                   push ecx
50                   push eax
50                   push eax
… …
8B 2B                mov ebp, [ebx]
```

a bit-pattern in Chernobyl/CIH virus

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

Malwares Are Obfuscating

- Obfuscating Methods
  - dead code insertion
  - code transposition
  - register reassignment
  - instruction substitution

- More powerful generation: polymorphic virus
  - Morphs every time it infects another program

- Failure of famous commercial detectors
  - Norton®, McAfee®, Command® (Christodorescu03)

- So, new Semantics (Behavioral) approach is required
  - What they do will not change even after the obfuscations

Ideal Solution

- Can we decide the following?

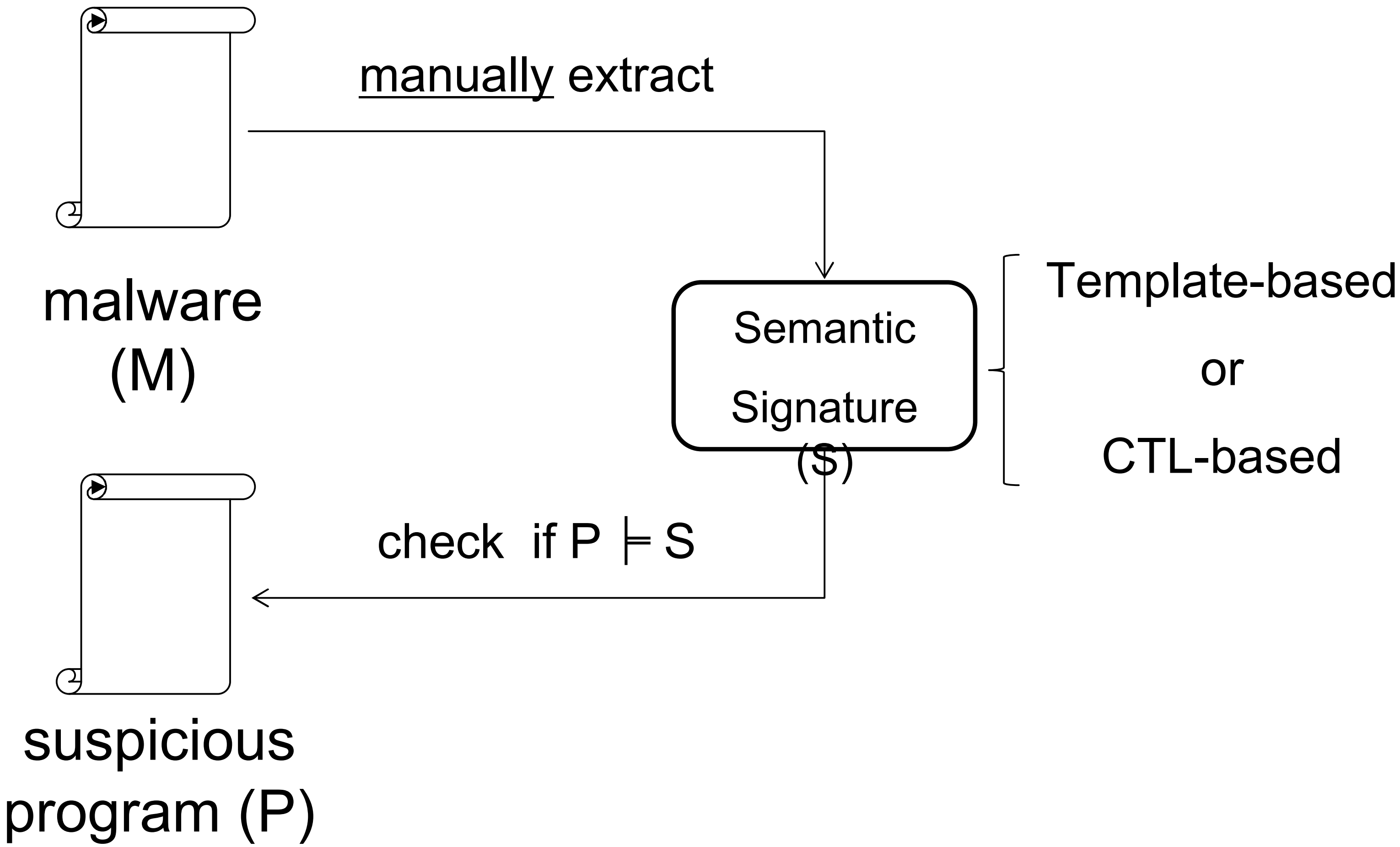
Two programs P and Q do the same things

- Unfortunately, no (Rice theorem)

Whether a program's behavior satisfies a nontrivial property or not ➔ Undecidable

- So, we have to find meaningful sub-domain of the problem or a inaccurate but sound solution

Current Approaches



Template-based Approach

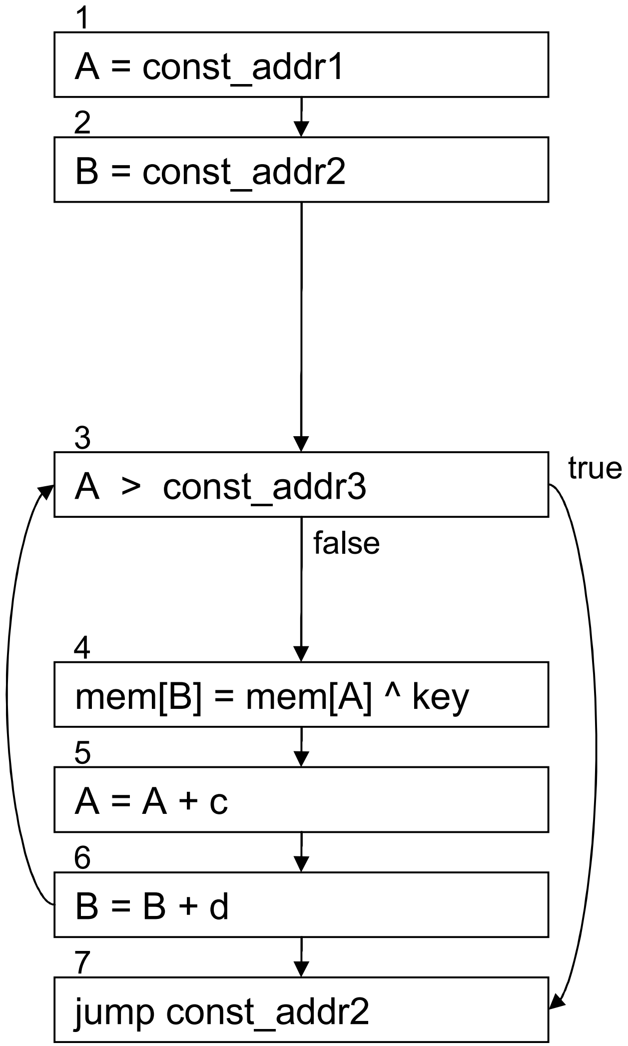
- Christodorescu et al, 2005

- Template
  - A Simple programming language with instruction, variable and symbolic constants
  - Expressive enough to describe the behavior of a binary program
    - Assignment to variable
    - Memory read/write
    - Unary/Binary operations
    - Jump, Branch

- Def-use path for a template variable A
  - A possible execution path (N_D, N_1, N_2, N_3, …, N_k, N_U) s.t.
    - A is defined in node N_D and used in N_U
    - N_1, N_2, N_3, …, N_k do not redefine A

Template-based Approach

e.g.



- Def-use path for A
  - (1,2,3)
  - (1,2,3,4)
  - (5,6,3)
  - (5,6,3,4)

Template-based Approach

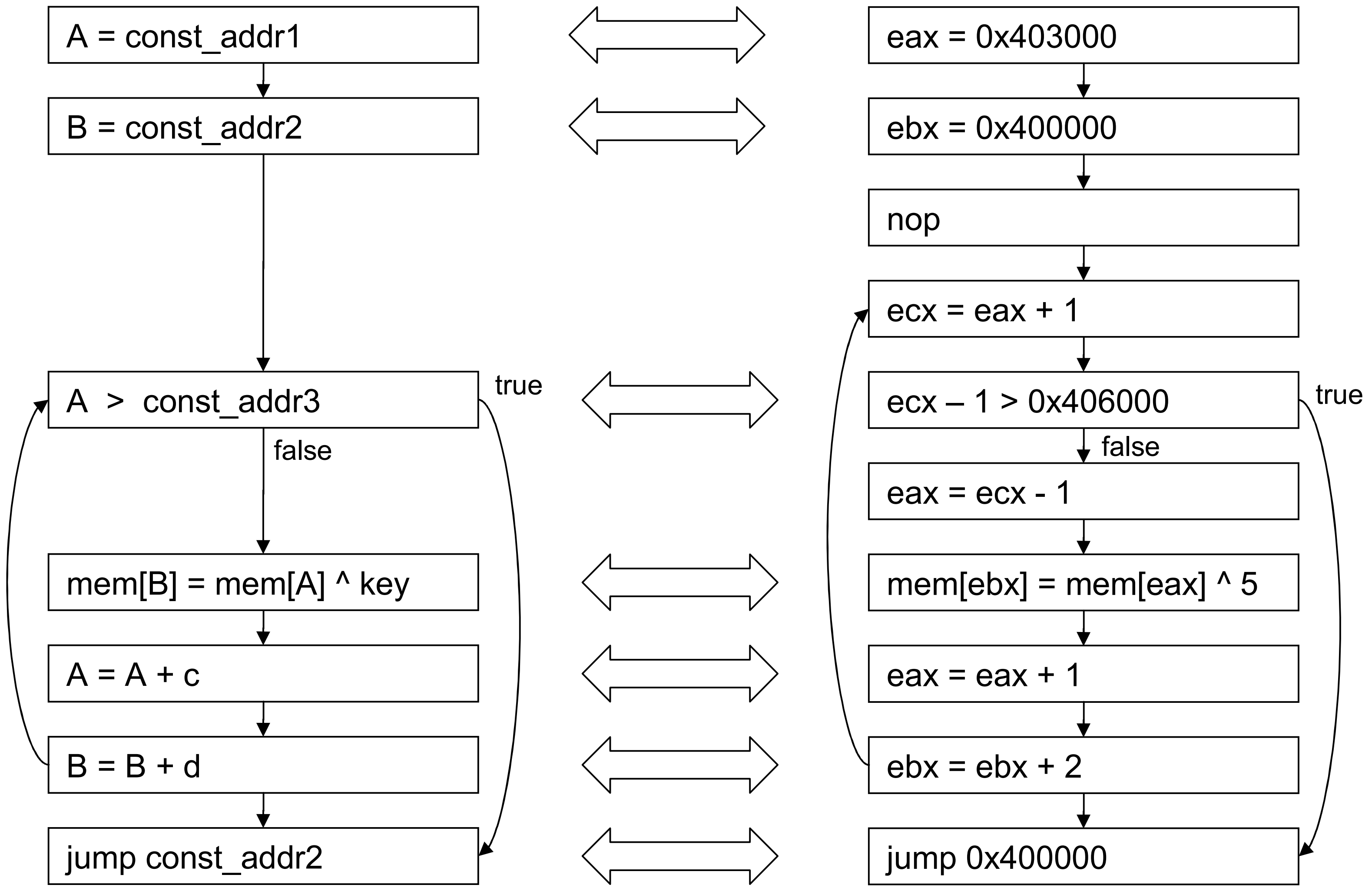
- Definition of P $\models$ T
  - Cond. 1 : If template updates a memory location, program also updates there with the same value
  - Cond. 2 : Program's event sequence subsumes Template's
  - Cond. 3 : If template ends at updated memory area, program does too

Template-based Approach

- The algorithm
  - First, tries to unify template and program nodes
  - Then, check value preservation on def-use paths
    - Two program expressions unified to a template variable on the ends of a path have the same value
    - By Decision procedure; identifying actual nop, symbolic execution, theorem proving

- They prove this is sound to prove that ‘P $\models$ T’

Template-based Approach



CTL-based (model checking) Approach

- Kinder et al, 2005
- See a binary executable as a Kripke system, and use CTL variation to describe and check malicious behavior
- Kripke system : finite state automata labeled with propositions
  - triple <S,R,L> and set of propositions P
  - S : set of states
  - R : subset of S * S, transitions
  - L : S → 2^P , called labeling function
    - If p is in L(S), then we say ‘p is satisfied in S’

CTL-based (model checking) Approach

- Kinder et al, 2005
- View a binary executable as a Kripke system
- Kripke system :
  - finite state automata labeled with sets of predicates
  - triple <S,R,L> and a set of predicates P
    - S : states
    - R : subset of S*S, transitions
    - L : S → 2^P, labeling function
  - If a predicate $p(r_1,r_2,\ldots,r_n) \in L(s)$, then we say '$p(r_1,r_2,\ldots,r_n)$ is satisfied in state s'

CTL-based (model checking) Approach

- View a binary executable as a Kripke system
  - S : instructions
  - R : control flows
  - P : obtained from each instruction
    - Instruction opcode → predicate name
    - Instruction operand → predicate operand
    - e.g. **cmp ebx, [bp-4]** → **cmp(ebx, [bp-4])**

- Then, use CTL to describe and check malicious behavior
  - Use CTL temporal operators: A, E, X, F, G, U
  - Allow quantifiers ∀, ∃ for predicate operands

$\mathbf{EF}(mov(eax,937)) \wedge \mathbf{AF}(push(eax)) \vee$
$\mathbf{EF}(mov(ebx,937)) \wedge \mathbf{AF}(push(ebx)) \vee$
$\mathbf{EF}(mov(ecx,937)) \wedge \mathbf{AF}(push(ecx))$

⇒

$\exists r.\ \mathbf{EF}(mov(r,937)) \wedge \mathbf{AF}(push(r))$

CTL-based (model checking) Approach

- Examples
  - "Set a register to 0 and push this onto the stack in the next instruction"

    → $\exists$ r. **EF**(mov(r,0) $\wedge$ **EX**(push(r))

  - "Set a register to 0 and push this onto the stack in the future instruction"

    → $\exists$ r. **EF**(mov(r,0) $\wedge$ **EF**(push(r))

  - "In the above, disallow intermediate update of r until push"

    → $\exists$ r. **EF**(mov(r,0) $\wedge$ **E**($\neg\exists$ t.mov(r,t) **U** push(r)))

CTL-based (model checking) Approach

$$
\begin{array}{l}
\exists L_m \exists L_c \exists v_{File}(\\
\quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0(\\
\quad\quad \mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0,t) \vee \texttt{lea}(r_0,t)))\mathbf{U}\#\mathrm{loc}(L_0))\wedge \\
\quad\quad \mathbf{EF}(\texttt{mov}(r_1, \texttt{0}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_1,t) \vee \texttt{lea}(r_1,t)))\mathbf{U}\#\mathrm{loc}(L_1))\wedge \\
\quad\quad \mathbf{EF}(\texttt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
\quad\quad\quad \mathbf{U}(\texttt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
\quad\quad\quad\quad \mathbf{U}(\texttt{push}(r_1) \wedge \#\mathrm{loc}(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
\quad\quad\quad\quad\quad \mathbf{U}(\texttt{call}(\texttt{GetModuleFileNameA}) \wedge \#\mathrm{loc}(L_m))))) \\
\quad\quad) \\
\quad \wedge(\exists r_0 \exists L_0(\\
\quad\quad \mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0,t) \vee \texttt{lea}(r_0,t)))\mathbf{U}\#\mathrm{loc}(L_0))\wedge \\
\quad\quad \mathbf{EF}(\texttt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
\quad\quad\quad \mathbf{U}(\texttt{call}(\texttt{CopyFileA}) \wedge \#\mathrm{loc}(L_c))) \\
\quad\quad)) \\
\quad \wedge\mathbf{EF}(\#\mathrm{loc}(L_m) \wedge \mathbf{EF}\#\mathrm{loc}(L_c)) \\
)
\end{array}
$$

<A CTL formula corresponding to Klez.h infection routine>

Our Interest and Direction

- Our Interest
  - Checking arbitrary program semantics
  - What is not my interest
    - specific behaviors of some malwares

- Our Direction
  - Further improvement of existing solutions?
    - Flaws in the definition of template behavior containment
    - More effective and concise way of expressing specification and dependency in model checking approach
    - Optimize. Improve performance and scalability of existing solutions

  - Automatic or computer-aided semantic signature extraction

  - Or …

Our Hope?

- Restricting the domain and find meaningful sub-problem
  - E.g. can we state program semantics elegantly if we use well-designed programming language?
    - Like the case of Termination Analysis
  - Automatic semantics extraction

Quick Sort Algorithm → Bubble Sort Algorithm →

<u>Pre-condition:</u>

Inputs $<a_1,a_2,a_3,\ldots a_n>$ are integers

<u>Post-condition:</u>

Outputs $<b_1,b_2,b_3,\ldots b_n>$ are sorted permutation of input

Semantics extracted automatically

So, we can safely conclude they do the same thing

Thank you

Q & A

CTL-based (model checking) Approach

- A trick to express dependency or order
  - For a label L, use #loc(L) predicates

- Example
  - "Call a function that takes two parameters, where the second one takes 0"
    - → $\exists L. \exists r2.\ (\ \mathbf{EF}(mov(r2,0)\ \wedge\ \mathbf{EF}\#loc(L))\ \wedge$
      $\exists r1.\ \mathbf{EF}(push(r1)\ \wedge \mathbf{EF}(push(r2)\ \wedge\ \#loc(L)\ \wedge \mathbf{EF}(call(func)))))$