Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification

Matt Webster and Grant Malcolm
Department of Computer Science
University of Liverpool, UK

17th Annual EICAR Conference
May 2008

Structure of the Presentation

- Introduction
  - Formal software specification in Maude
- Formal detection of metamorphic viruses
  - Dynamic analysis
  - Static analysis – equivalence in context
- Formal detection of virtualization-based viruses
- Conclusion

Formal Software Specification in Maude

- Maude consists of two parts:
  - Software specification language
    - Algebraic
  - Term rewriting engine
    - Equational and Rewriting logics
- Maude has been used to specify many different languages
  - Java, Prolog, Scheme...
  - ... Intel 64 assembly language
- Maude is formal... therefore we can use it to prove program equivalence

A Maude Specification of Intel 64

- Our specification is based on *store semantics*
- Syntax of instructions

```
MOV_,_ : Variable Expression -> Instruction
```

- Semantics of instructions

```
S ; MOV V,E [[V]] = S[[E]]

S ; MOV V1,E [[V2]] = S[[V2]] if V1 =/= V2
```

- So far we have done this for MOV, ADD, SUB, OR, AND, XOR, TEST, PUSH, POP, NOP
  - In principle, this subset can be extended further

Dynamic Analysis

- We can use the Maude term rewriting engine to successively apply equations
  - The result gives us the final value of some variable

```
      s ; MOV eax,0 ; MOV ebx, eax [[ebx]
==> s ; MOV eax,0 [[eax]]
==> s [[0]] ==> 0
```

- Equations used:
  - `S ; MOV V,E [[V]] = S[[E]]`
  - `S ; MOV V1,E [[V2]] = S[[V2]] if V1 =/= V2`
- We can do the same for sequences of instructions
  - Effectively, we have an interpreter for MOV
  - The same can be done for the rest of Intel 64

Dynamic Analysis in Practice

- We can do dynamic analysis using Maude to detect metamorphic viruses (Webster & Malcolm, 2006)
- Win95/Bistro

a

```
push ebp
mov ebp, esp
```

b

```
push ebp
push esp
pop ebp
```

- Perform equational rewrites using Maude

```
Maude> reduce s ; a [[stack]] is s ; b [[stack]].
result: true

Maude> reduce s ; a [[ebp]] is s ; b [[ebp]] .
result Bool: true
```

- Therefore these fragments are equivalent*

* We have restricted attention to esp, ebp and the stack for the sake of simplicity

A Problem with Equivalence-based Detection

- Metamorphic viruses need not rewrite themselves with equivalent code, e.g., Win9x.Zmorph.A

```
mov edi, 2580774443
mov ebx, 467750807
sub ebx, 1745609157
sub edi, 150468176
xor ebx, 875205167
push edi
xor edi, 3761393434
push ebx
push edi
```

```
mov ebx, 535699961
mov edx, 1490897411
xor ebx, 2402657826
mov ecx, 3802877865
xor edx, 3743593982
add ecx, 2386458904
push ebx
push edx
push ecx
```

- After executing both fragments, the stack and the instruction pointer have the same values. However, registers edi, ebx, ecx and edx differ
- We call this condition *semi-equivalence*

Equivalence in Context

- Win9x.Zmorph.A

Semi-equivalent code

```
mov edi, 2580774443          mov ebx, 535699961
mov ebx, 467750807           mov edx, 1490897411
sub ebx, 1745609157          xor ebx, 2402657826
sub edi, 150468176           mov ecx, 3802877865
xor ebx, 875205167           xor edx, 3743593982
push edi                     add ecx, 2386458904
xor edi, 3761393434          push ebx
push ebx                     push edx
push edi                     push ecx

mov edi, 0                   mov edi, 0
mov ebx, 0                   mov ebx, 0
mov ecx, 0                   mov ecx, 0
mov edx, 0                   mov edx, 0
```

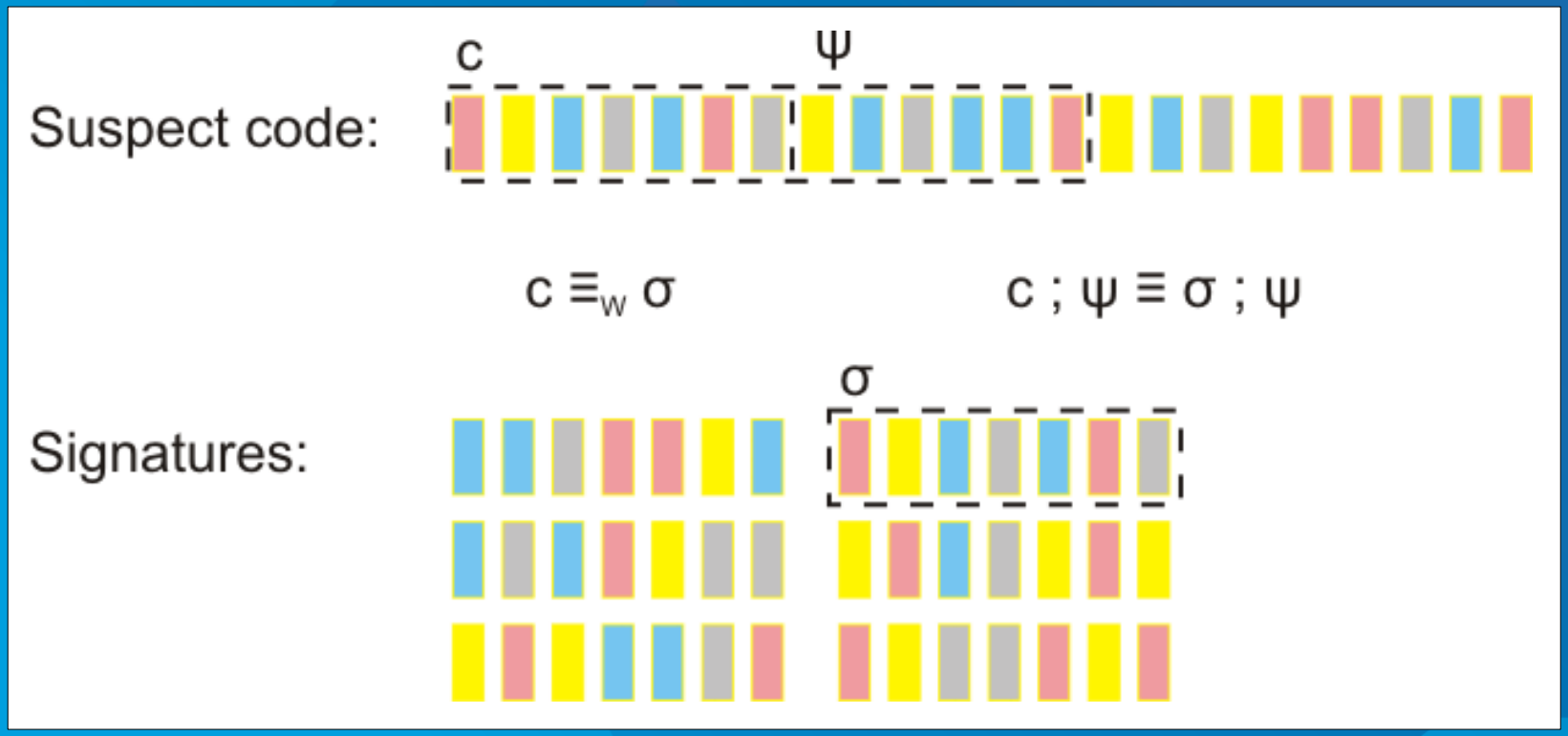
- After executing, all variables have the same values
- This is called *equivalence in context*

Equivalence in Context (2)

- There may be other conditions under which equivalence in context applies.
- In general:
  - If p_1 and p_2 are semi-equivalent instruction sequences...
  - ... and they are both followed by p...
  - ... and p's behaviour is not affected by the unequal variables in p_1 and p_2...
  - ... and p overwrites all the unequal variables...
  - Then p_1 and p_2 are equivalent in context of p.
- This is the Equivalence in Context Theorem

Equivalence in Context (3)

- Equivalence in Context can be applied to detection



Equivalence in Context (4)

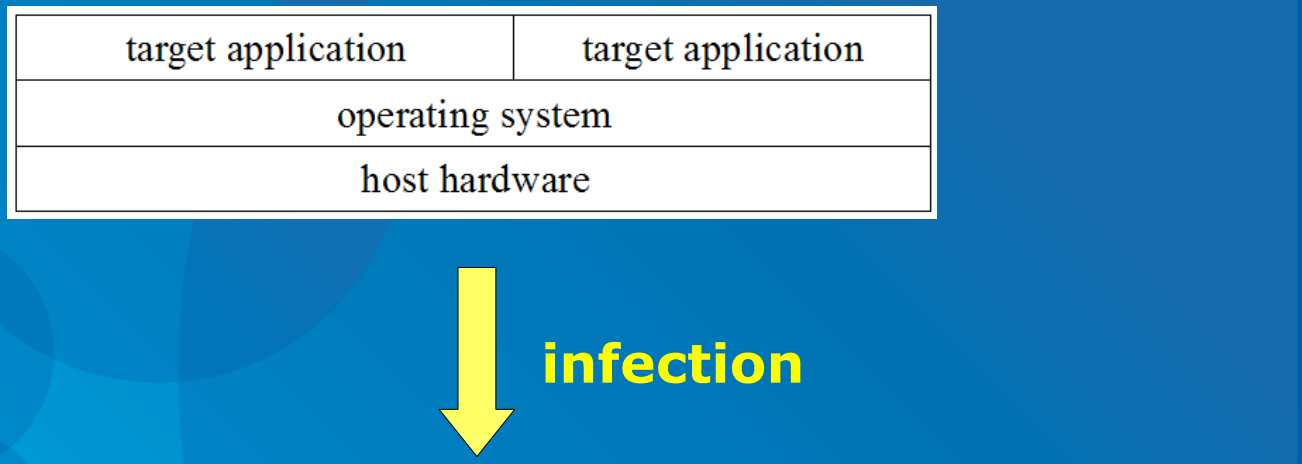
- The Equivalence in Context Theorem holds for all instruction sequences
  - Q. How does the Maude specification of Intel 64 help?
  - A. We can use the Maude specification to determine:
    - which variables affect the behaviour of an instruction
    - which variables are affected by an instruction
  - Therefore, the Maude specification of Intel 64 is useful for applying equivalence in context

Detection of Virtualization-based Malware

- Previously, we used the Maude specification of Intel 64 for dynamic analysis
- However, we can also use it to generate code automatically...
  - ...according to some specification
- To do this, we use Maude's built-in search functionality
- This can be applied to detection of virtualization-based malware

Virtualization-based Malware

- Virtual machine-based rootkits (VMBRs) (King et al, 2006)



Detecting Virtualization-based Malware

- Programs such as Blue Pill can detect VMBRs
  - Use the SIDT instruction (Rutkowska, 2004)
    - Returns the contents of the interrupt descriptor table
    - The IDT differs during virtualization
- However, VMBRs can use countermeasures
  - Detect when Blue Pill is loaded
  - Breakpoint on the SIDT instruction
  - Emulate SIDT to hide virtualization from Blue Pill
- What if we generate SIDT at run time?
  - Detection of Blue Pill/SIDT not possible
  - Detection of malware by SIDT will still work

Detecting Virtualization-based Malware (2)

- We can use the Maude specification of Intel 64
  - Generate new "variants" of Blue Pill automatically
- Q. Why not just use a metamorphic engine?
  - The Maude specification of Intel 64 is formal
    - Each generated variant is automatically verified formally
  - Very little programming required
  - Metamorphic engines are likely to be buggy

Detecting Virtualization-based Malware (3)

- Proof of concept system

```
rl [1] : S[[eax]] => S ; mov ebx, "sidt" [[eax]] .
rl [2] : S[[eax]] => S ; mov eax, ebx [[eax]] .
rl [3] : S[[eax]] => S ; mov ecx, ebx [[eax]] .
rl [4] : S[[eax]] => S ; mov eax, ecx [[eax]] .
```

Let the end condition be s[[eax]] = "sidt"

Then, apply any of the following to reach the end condition from s[[eax]]:

(1,2), (1,2,3), (1,2,3,4), (1,3,4), (1,3,3,4), (1,3, ... ,3,4), ...

- Produces 1000 different programs in ~0.36 seconds

Future Work

- Detection of metamorphic viruses
  - Specify a larger subset of Intel 64
  - Investigate equivalence in context
    - Loops
    - Conditionals
- Detection of virtualization-based malware
  - Scale up the proof-of-concept system
    - Produce programs that generate SIDT on the fly, and execute it

Conclusion

- Intel 64 specification in Maude
  - Detection of metamorphic viruses
    - Dynamic analysis
    - Static analysis (Equivalence in Context)
  - Detection of virtualization-based malware
    - Automatic generation of formally-verified “Blue Pill” programs

End of Presentation

- Any questions?