

Efficient Malware Detection using Model-Checking

Tayssir Touili

LIPN, CNRS & Univ. Paris 13

Motivation: Malware Detection

- The number of new malware exceeds **75 million** by the end of 2011, and is still increasing.
- The number of malware that produced incidents in 2010 is more than **1.5 billion**.
- The **worm MyDoom slowed down global internet access** by **10%** in 2004.
- Authorities investigating the 2008 **crash of Spanair flight 5022** have discovered a central computer system used to monitor technical problems in the aircraft **was infected with malware**

Motivation: Malware Detection

- The number of new malware exceeds **75 million** by the end of 2011, and is still increasing.
- The number of malware that produced incidents in 2010 is more than **1.5 billion**.
- The worm MyDoom slowed down global internet access by **10%** in 2004.
- Authorities have discovered a

**Malware detection is
important!!**

Limitations of classic anti-virus techniques

- **Signature (pattern) matching:** Every known malware has one signature

Limitations of classic anti-virus techniques

- **Signature (pattern) matching:** Every known malware has one signature
 - Easy to get around
 - New variants of viruses with the same behavior cannot be detected by these techniques
 - Nop insertion, code reordering, variable renaming, etc
 - Virus writers frequently update their viruses to make them undetectable

Limitations of classic anti-virus techniques

- **Signature (pattern) matching:** Every known malware has one signature
 - Easy to get around
 - New variants of viruses with the same behavior cannot be detected by these techniques
 - Nop insertion, code reordering, variable renaming, etc
 - Virus writers frequently update their viruses to make them undetectable
- **Code emulation:** Executes binary code in a virtual environment

Limitations of classic anti-virus techniques

- **Signature (pattern) matching:** Every known malware has one signature
 - Easy to get around
 - New variants of viruses with the same behavior cannot be detected by these techniques
 - Nop insertion, code reordering, variable renaming, etc
 - Virus writers frequently update their viruses to make them undetectable
- **Code emulation:** Executes binary code in a virtual environment
 - Checks program's behavior only in a limited time interval

Limitations of classic anti-virus techniques

- **Signature (pattern) matching:** Every malware has one signature

Solution:

Check the behavior (not the syntax) of the program without executing it

- Virus writers use obfuscation to make malware undetectable
- **Code emulation:** Execute binary code in a virtual environment
 - Checks program's behavior only in a limited time interval

Limitations of classic anti-virus techniques

- **Signature (pattern) matching:** Every malware has one signature

Solution:

Check the behavior (not the syntax) of the program without executing it

- Virus writers use obfuscation to make malware undetectable
- **Code emulation:** Executes binary code in a virtual environment
 - Checks program's behavior only in a limited time interval

Model Checking is a good candidate

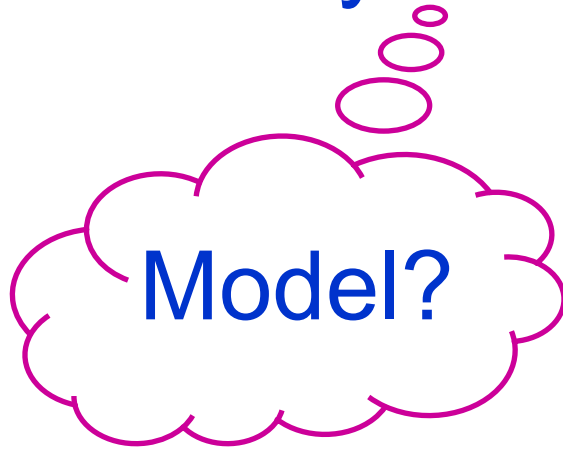
Goal: Model-checking for malware detection

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



Model?



Specification
formalism?

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



Model?



Specification formalism?

Existing works: use finite automata to model the programs

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



Model?



Specification formalism?

Existing works: use finite automata to model the programs



Stack?

Stack: important for malware detection

- To achieve their goal, malware have to call functions of the operating system
- Antiviruses determine malware by checking the calls to the operating systems.
- Virus writers try to hide these calls.

Stack: important for malware detection

- To achieve their goal, malware have to call functions of the operating system
- Antiviruses determine malware by checking the calls to the operating systems.
- Virus writers try to hide these calls.

```
L0 : call f
L1: ...
...
...
f : function f
```

```
L0 : push L1
L'0: jmp f
L1: ...
...
...
f : function f
```

Stack: important for malware detection

Important to analyse the program's stack

- To achieve their objectives, malware writers often use functions of the operating system to perform various tasks.
- Anti-virus software can detect these calls to the operating system.
- Virus writers try to hide these calls.

```
L0 : call f
L1: ...
...
...
f : function f
```

```
L0 : push L1
L'0: jmp f
L1: ...
...
...
f : function f
```

Stack: important for malware detection

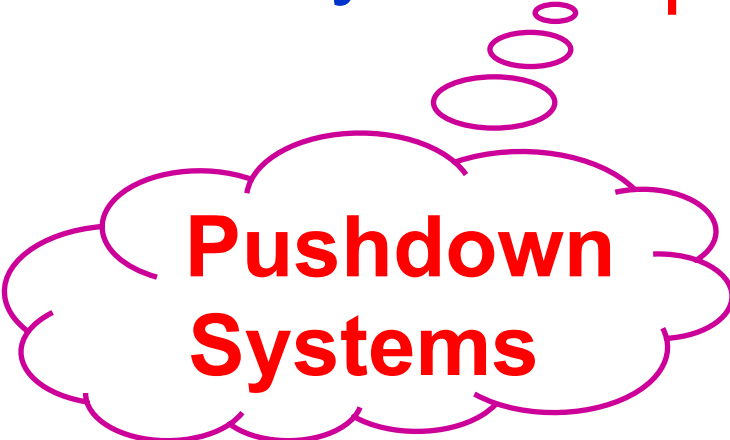
Important to analyse the program's stack

Solution:

Use pushdown systems to model programs

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?




**Pushdown
Systems**



Specification
formalism?

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



The diagram consists of two thought bubbles connected by a line. The left bubble contains the text 'Pushdown Systems' in red. The right bubble contains the text 'Specification formalism?' in blue. Above the left bubble are three small circles, and above the right bubble are two small circles, suggesting a flow or connection between the two concepts.

**Pushdown
Systems**

**Specification
formalism?**

Specification of malicious behaviors?

Example: fragment of email worm Avron

Call the API `GetModuleHandleA` with `0` as parameter.
This returns the entry address of its own executable.
Copy itself to other locations.

```
mov eax, 0  
push eax  
call GetModuleHandleA
```

Specification of malicious behaviors?

Example: fragment of email worm Avron

Call the API `GetModuleHandleA` with `0` as parameter.
This returns the entry address of its own executable.
Copy itself to other locations.

```
mov eax, 0  
push eax  
call GetModuleHandleA
```

How to describe this specification?

Specification of malicious behaviors?

Example: fragment of email worm Avron

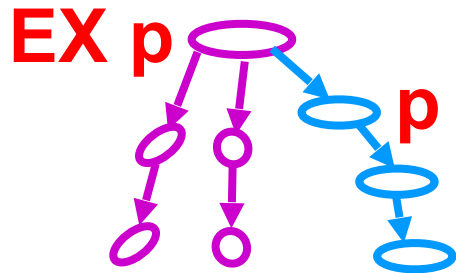
```
mov eax, 0  
push eax  
call GetModuleHandleA
```

In CTL (Branching-time temporal logic) :

$\text{mov}(\text{eax}, 0) \wedge \mathbf{EX} (\text{push}(\text{eax}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA})$

Specification of malicious behaviors?

Example: fragment of email worm Avron



```
mov eax, 0  
push eax  
call GetModuleHandleA
```

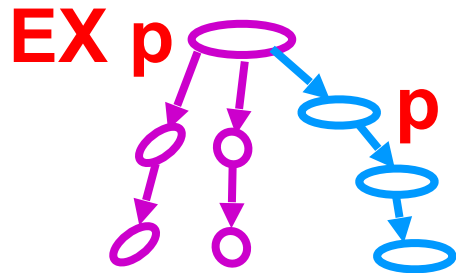
In CTL (Branching-time temporal logic) :

$\text{mov}(\text{eax}, 0) \wedge \mathbf{EX} (\text{push}(\text{eax}) \wedge \mathbf{EX} \text{ call GetModuleHandleA})$

EX p : there is a path where p holds at the next state

Specification of malicious behaviors?

Example: fragment of email worm Avron



```
mov eax, 0
push eax
call GetModuleHandleA
```

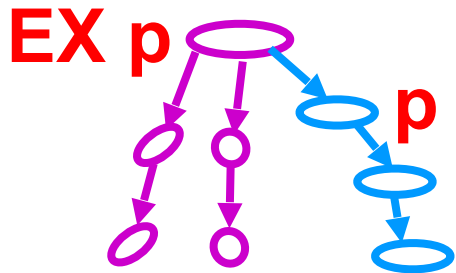
In CTL (Branching-time temporal logic) :

$$\begin{aligned} & \text{mov}(\text{eax}, 0) \wedge \mathbf{EX} (\text{push}(\text{eax}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA}) \\ & \vee \\ & \text{mov}(\text{ebx}, 0) \wedge \mathbf{EX} (\text{push}(\text{ebx}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA}) \\ & \vee \\ & \text{mov}(\text{ecx}, 0) \wedge \mathbf{EX} (\text{push}(\text{ecx}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA}) \\ & \vee \dots \text{all the other registers} \end{aligned}$$

EX p: there is a path where **p** holds at the next state

Specification of malicious behaviors?

Example: fragment of email worm Avron



```
mov eax, 0
push eax
call GetModuleHandleA
```

In **CTL** (Branching-time temporal logic) : **Huge!**

$\text{mov}(\text{eax}, 0) \wedge \mathbf{EX} (\text{push}(\text{eax}) \wedge \mathbf{EX} \text{ call GetModuleHandleA})$

\vee

$\text{mov}(\text{ebx}, 0) \wedge \mathbf{EX} (\text{push}(\text{ebx}) \wedge \mathbf{EX} \text{ call GetModuleHandleA})$

\vee

$\text{mov}(\text{ecx}, 0) \wedge \mathbf{EX} (\text{push}(\text{ecx}) \wedge \mathbf{EX} \text{ call GetModuleHandleA})$

\vee all the other registers

EX p: there is a path where *p* holds at the next state

Specification of malicious behaviors?

Example: fragment of email worm Avron

CTPL = CTL +
variables + \exists, \forall

```
mov eax, 0  
push eax  
call GetModuleHandleA
```

In CTL:

mov(**eax**,0) \wedge **EX** (push(**eax**) \wedge **EX** call **GetModuleHandleA**)

\vee

mov(**ebx**,0) \wedge **EX** (push(**ebx**) \wedge **EX** call **GetModuleHandleA**)

\vee

mov(**ecx**,0) \wedge **EX** (push(**ecx**) \wedge **EX** call **GetModuleHandleA**)

\vee **all the other registers**

Specification of malicious behaviors?

Example: fragment of email worm Avron

CTPL = CTL +
variables + \exists, \forall

```
mov eax, 0  
push eax  
call GetModuleHandleA
```

In CTL:

$\text{mov}(\text{eax}, 0) \wedge \mathbf{EX} (\text{push}(\text{eax}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA})$

\vee

$\text{mov}(\text{ebx}, 0) \wedge \mathbf{EX} (\text{push}(\text{ebx}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA})$

\vee

$\text{mov}(\text{ecx}, 0) \wedge \mathbf{EX} (\text{push}(\text{ecx}) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA})$

$\vee \dots \dots$ all the other registers

In CTPL:

$\exists r (\text{mov}(r, 0) \wedge \mathbf{EX} (\text{push}(r) \wedge \mathbf{EX} \text{ call } \text{GetModuleHandleA}))$

Specification of malicious behaviors?

Example: fragment of email worm Avron

CTPL = CTL +
variables + \exists

mov(ecx, 0)

mov(ecx, A)

In CTL:

**CTPL cannot describe the stack:
needed for malicious behaviors
description**

mov(ecx, 0)

mov(ecx, 0)

mov(ecx, A)

the other registers

In CTPL:

$\exists r$ (mov(r , 0) \wedge EX (push(r) \wedge EX call GetModu A))



Specification of malicious behaviors?

Example: fragment of email worm Avron

Call the API `GetModuleHandleA`
with `0` as parameter.
This returns the entry address of its
own executable.
Copy itself to other locations.

```
mov eax, 0  
push eax  
call GetModuleHandleA
```

In CTPL:

$\exists r \text{ (mov}(r, 0) \wedge \mathbf{EX} \text{ (push}(r) \wedge \mathbf{EX} \text{ call GetModuleHandleA))}$

Specification of malicious behaviors?

Example: fragment of email worm Avron

Call the API `GetModuleHandleA` with 0 as parameter.

This returns the entry address of its own executable.

Copy itself to other locations.

```
mov eax, 0
push ebx
pop ebx
push eax
call GetModuleHandleA
```

In CTPL:

$\exists r \text{ (mov}(r, 0) \wedge \mathbf{EX} \text{ (push}(r) \wedge \mathbf{EX} \text{ call GetModuleHandleA))}$

Specification of malicious behaviors?

Example: fragment of email worm Avron

Call the API `GetModuleHandleA`
with `0` as parameter.
This returns the entry address of its
own executable.
Copy itself to other locations.

```
mov eax, 0  
push ebx  
pop ebx  
push eax  
call GetModuleHandleA
```

In CTPL:

$\exists r \text{ (mov}(r, 0) \wedge \mathbf{EX} \text{ (push}(r) \wedge \mathbf{EX} \text{ call GetModuleHandleA))}$

Our solution: Consider predicates over the stack

Specification of malicious behaviors?

Example: fragment of email worm Avron

Call the API `GetModuleHandleA`
with `0` as parameter.
This returns the entry address of its
own executable.
Copy itself to other locations.

```
mov eax, 0  
push ebx  
pop ebx  
push eax  
call GetModuleHandleA
```

In CTPL:

$\exists r \text{ (mov}(r, 0) \wedge \mathbf{EX} \text{ (push}(r) \wedge \mathbf{EX} \text{ call GetModuleHandleA))}$

Our solution: Consider predicates over the stack

In SCTPL:

$\mathbf{EF} \text{ (call GetModuleHandleA } \wedge \text{ (head_stack = 0))}$

EF p : there is a path where p holds in the future

Expressing Obfuscated Calls in SCTPL

L0 : call f

L: ...

...

...

f : function f

L0 : push L

L'0: jmp f

L: ...

...

...

f : function f

Expressing Obfuscated Calls in SCTPL

```
L0 : call f
L: ...
...
...
f : function f
```

```
L0 : push L
L'0: jmp f
L: ...
...
...
f : function f
```

$$\exists L \left(\mathbf{E} \ ! \left(\exists f \text{ call}(f) \wedge \mathbf{AX} \text{ (head_stack=L)} \right) \right. \\ \left. \mathbf{U} \text{ (ret} \wedge \text{(head_stack= L))} \right)$$

Expressing Obfuscated Calls in SCTPL

```
L0 : call f  
L: ...  
...  
...  
f : function f
```

```
L0 : push L1  
L'0: jmp f  
L: ...  
...  
...  
f : function f
```

L is not a return address of a function call

$$\exists L \left(\mathbf{E} \left[\overbrace{!(\exists f \text{ call}(f) \wedge \mathbf{AX}(\text{head_stack}=L))} \right] \right. \\ \left. \mathbf{U} \left(\text{ret} \wedge (\text{head_stack}=L) \right) \right)$$

Expressing Obfuscated returns in SCTPL

```
L0 : call f
a : ...
...
f: ...
...
pop eax
jmp eax
```

$$\mathbf{AG} \left(\forall f \forall a \left(\underbrace{(call(f) \wedge \mathbf{AX}_{a}^{h_s=})}_{a \text{ is a return address of a procedure call}} \implies \mathbf{AF} ! (ret \wedge \mathbf{h_s=}_a) \right) \right)$$

a is a return address of a procedure call

h_s : head-stack

Expressing Appending Viruses in SCTPL

An appending virus append itself at the end of the host file
The virus has to compute its address in memory

```
L0 : call f  
a :  
...  
f: pop eax
```

$$\mathbf{AG} \left(\underbrace{\forall f \forall a \left((call(f) \wedge \mathbf{AX}_{h_s = a}} \right)}_{\substack{a \text{ is a return address} \\ \text{of a procedure call}}} \implies \mathbf{AF} \neg r(pop(r) \wedge \mathbf{h_s}_{=a}) \right)$$

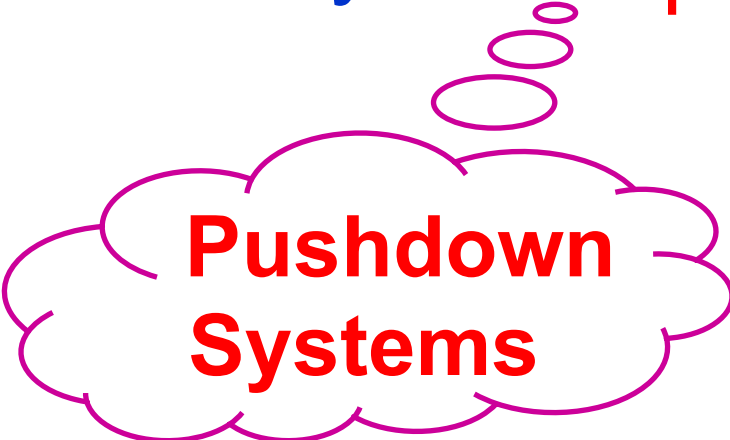
h_s : head-stack

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



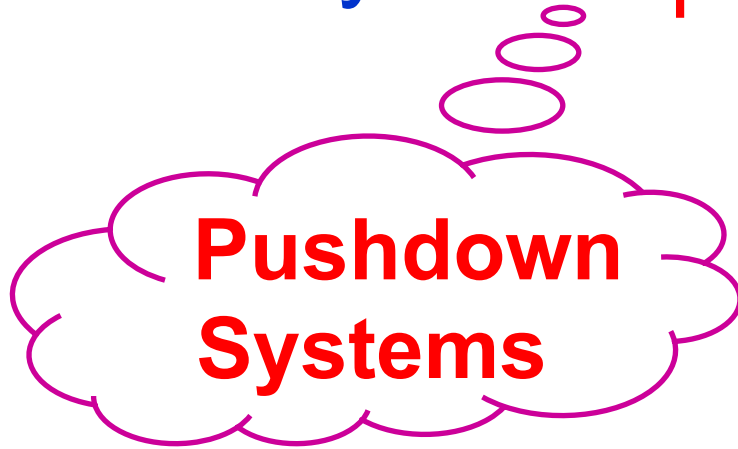
**Pushdown
Systems**



SCTPL

Goal: Model-checking for malware detection

Binary code \models Malicious behavior ?



Pushdown System \models SCTPL ?

SCTPL model-checking for Pushdown Systems

Non trivial: stack can be unbounded

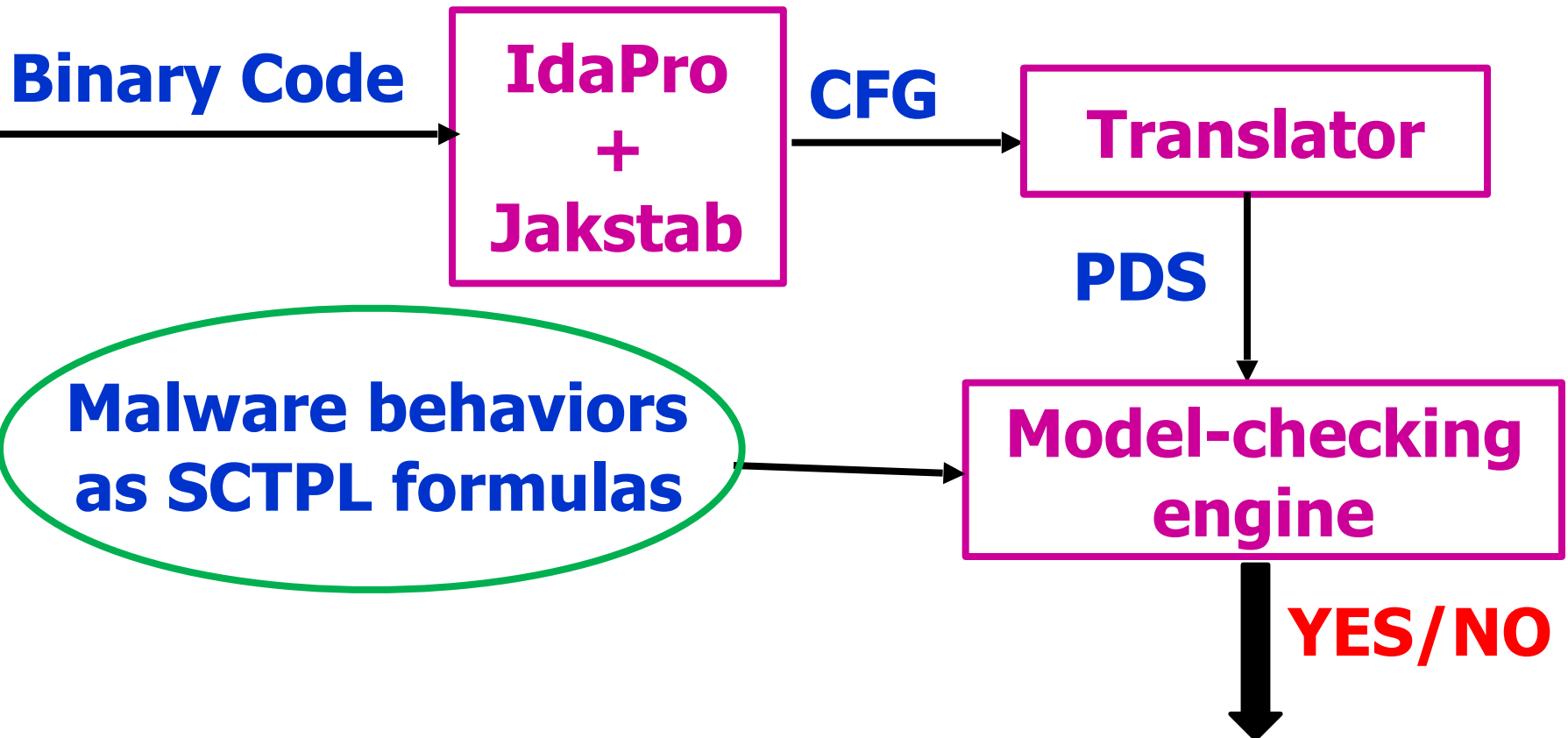
SCTPL model-checking for Pushdown Systems

Non trivial: stack can be unbounded

Theorem: Given a Pushdown System P and a SCTPL formula φ , whether P satisfies φ can be effectively decided.

Implementation

We implemented our techniques in a tool for virus detection



Experiments of PoMMADe

1. Our tool was able to detect more than 800 malwares
2. We checked 400 real benign programs from Windows XP system. Benign programs are proved benign with only three false positives.
3. Our tool was able to detect all the 200 new malwares generated by two malware creators
4. Analyze the Flame malware that was not detected for more than 5 years by any anti-virus

Our tool vs. known anti-viruses

NGVCK and VCL32 malware generators

1. generate 200 new malwares
2. the best malware generators
3. generate complex malwares

Generator	No. Of Variants	PO MM ADE	Avira	Kaspersky	Avast	Qihoo 360	McAfee	AVG	BitDefender	Eset Nod32	F-Secure	Norton	Panda	Trend Micro
NGVCK	100	100%	0%	23%	18%	68%	100%	11%	97%	81%	0%	46%	0%	0%
VCL32	100	100%	0%	2%	100%	99%	0%	100%	100%	76%	0%	30%	0%	0%

Analyze The Flame Malware

Flame is being used for targeted cyber espionage in Middle Eastern countries.

It can

- 1.sniff the network traffic
- 2.take screenshots
- 3.record audio conversations
- 4.intercept the keyboard
- 5.and so on

It was not detected by any anti-virus for 5 years

Analyze The Flame Malware

Flame is being used for targeted cyber espionage in Middle Eastern countries.

It can

- 1.sniff the network traffic
- 2.take screenshots
- 3.record audio conversations
- 4.intercept the keyboard
- 5.and so on

It was not detected by any anti-virus for 5 years

Our tool can detect this malware Flame

Conclusion

- We introduced a new logic SCTPL to precisely specify malicious behaviors
- We proposed efficient SCTPL model-checking algorithms for pushdown systems.
- We implemented our techniques in a tool for malware detection: **POMMADE**
- **POMMADE** was able to detect more than **800** malwares, several of them cannot be detected by well-known anti-viruses, such as, Avast, Kaspersky, McAfee, Norton, Avira, etc

Questions?