On Static Malware Detection

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 i 10 is more than 1.5 billion.
- Th rm MyDoom slowed global int ss by 10% in 2004.
- Authoriti ve discovered a infec

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 there 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 there 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 there 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

- **Sig re (pattern) ma :** Every halware has one signatu
  - F
  - etected by th
  - p ertic
  - Virus writers nti her o make ndetectable
- **Code emulation:** Exec inary code virtual environment
  - Checks program’s beha r only in a limited time i terval

Solution:

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

Static Analysis and Model Checking are good candidates

Goal: Static Analysis and 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.

```
L0 : call f
L1: …
 …
…
f :  function f
```

```
L0 : push L1
L’0: jmp f
L1: …
 …
 …
f :  function f
```

Stack: important for malware detection

- To ac… …ctions
- …he calls to the oper… …
- Virus writers try to … e these ca…

Important to analyse the program's stack

L0 : c…

…

f : function f

f … nction f

Solution:

Use pushdown systems to model programs

Pushdown Systems

PDS = finite automaton + Stack

P=(P, Γ, Δ),

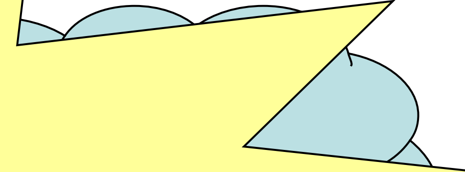
- P is a finite set of control states
- **Γ** is the stack alphabet
- $\Delta \subseteq (P\times\Gamma) \times (P\times\Gamma^*)$ is a finite set of transitions
- A configuration is a pair $\langle p,\omega\rangle \in P\times\mathbf{\Gamma}^*$
- If $\langle p, \alpha\rangle \rightarrow \langle p',\omega\rangle \in \Delta$, then, for every $u \in \Gamma^*$,

  $\langle p, \alpha u\rangle \Rightarrow \langle p',\omega u\rangle$

From Binary Codes to PDSs

Difficulty:

```
mov
call
```



It’s non-trival to get registers’ values

Computing Registers’ Values

We need an oracle that computes the values of the registers

```
mov eax, 1
dec eax
push eax
call GetModuleHandleA
```

eax’s value is 0

We use Jakstab [Kinder-Veith 2008] to implement the oracle

Jakstab (Java Toolkit for static analysis of binaries) does a kind of constant propagation to determine registers’ values

From Binary Codes to PDSs

l_1: mov eax, 1

l_2: dec eax

l_3: push eax

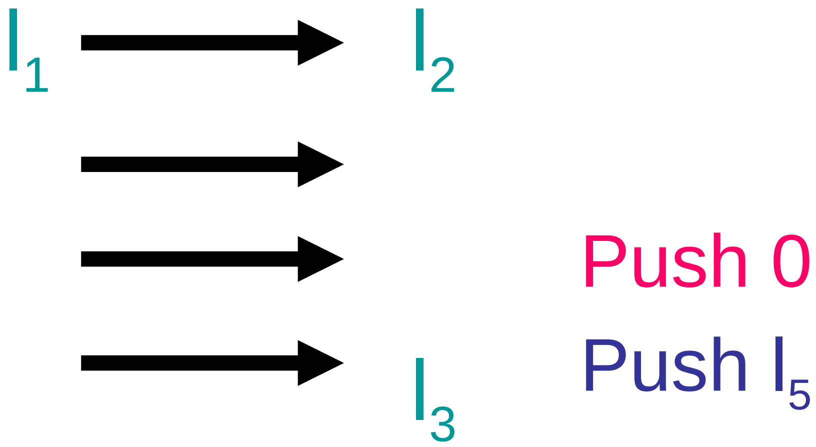
l_4: call GetModuleHandleA

l_5: ...

g_0= entry point of GetModuleHandeA

Control states of PDS = control points of program

Stack alphabet = return addresses+ registers' values



Malicious behaviors?

Binary code $\models$ Malicious behavior ?

✓

PDS

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

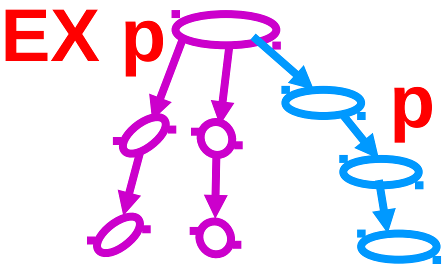
Call the API GetModuleHandleA with 0 as parameter.
This returns the entry address of its own executable.
Copy itself to other locations.

```
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
```

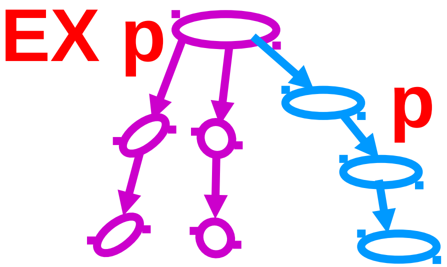
In CTL (Branching-time temporal logic) **:**

mov(eax,0)^**EX (**push(eax)^**EX** 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) **:**

mov(eax,0)^**EX (**push(eax)^**EX** call GetModuleHandleA**)**

˅

mov(ebx,0)^**EX (**push(ebx)^**EX** call GetModuleHandleA**)**

˅

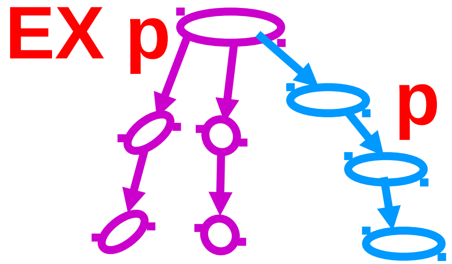
mov(ecx,0)^**EX (**push(ecx)^**EX** call GetModuleHandleA**)**

˅ ….. 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



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

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

Huge!

mov(eax,0)ˆ**EX (**push(eax)ˆ**EX** call GetModuleHandleA**)**

˅

mov(ebx,0)ˆ**EX (**push(ebx)ˆ**EX** call GetModuleHandleA**)**

˅

mov(ecx,0)ˆ**EX (**push(ecx)ˆ**EX** call GetModuleHandleA**)**

˅ ….. 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 + ∃, ∀

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

In CTL:

mov(eax,0)^**EX (**push(eax)^**EX** callGetModuleHandleA**)**

∨

mov(ebx,0)^**EX (**push(ebx)^**EX** callGetModuleHandleA**)**

∨

mov(ecx,0)^**EX (**push(ecx)^**EX** callGetModuleHandleA**)**

∨ ….. all the other registers

In CTPL:

∃ r **(**mov(r,0)^**EX (**push(r)^ **EX** call GetModuleHandleA**))**

Specification of malicious behaviors?

Example: fragment of email worm Avron

CTPL = CTL

iables + ∃

mo 0

A

In CTL:

CTPL cannot describe the stack: needed for malicious behaviors description

mov(

mov(ecx,0)^ **EX** odule A)

˅ .... e other regis rs

In CTPL:

∃ r (mov(r,0)^**EX** (push(r)^ **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:

∃ r **(**mov(r,0)^**EX (**push(r)^ **EX** 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:

∃ r **(**mov(r,0)^**EX (**push(r)^ **EX** … A**))**

the head of stack is 0

Our solution: Consider predic… …ck

In SCTPL:

EF (call GetModuleHandleA ^ $0\Gamma^*$ **)**

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

SCTPL Logic

$\phi ::= b \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{EX}\ \phi \mid \mathbf{E}[\phi\ \mathbf{U}\ \phi] \mid \mathbf{EG}\ \phi$

SCTPL Logic

$\phi ::= b(y_1, \ldots, y_n) \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{EX}\ \phi \mid \mathbf{E}[\phi\ \mathbf{U}\ \phi] \mid \mathbf{EG}\ \phi$

- $y \in$ **Y**, a set of variables over a finite domain **D**

SCTPL Logic

$\phi ::= b(y_1,\ldots,y_n) \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{EX}\ \phi \mid \mathbf{E}[\phi\mathbf{U}\phi] \mid \mathbf{EG}\ \phi \mid \exists \mathbf{y}\ \phi$

- $y \in$ **Y**, a set of variables over a finite domain **D**

SCTPL Logic

$\phi ::= b(y_1,\ldots,y_n) \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{EX}\ \phi \mid \mathbf{E}[\phi \mathbf{U} \phi] \mid \mathbf{EG}\ \phi \mid \exists \mathbf{y}\ \phi \mid e$

- $y \in \mathbf{Y}$, a set of variables over a finite domain **D**
- e is a regular expression over $\mathbf{Y} \cup \Gamma$

Expressing Obfuscated Calls in SCTPL

L_0: call f
L_1: …
…
…
f : function f

Normal function

Obfuscate the call

L_0: push L_1
L_2: jmp f
L_1: …
…
…
f unction f

cated function call

$L\Gamma^*$ = predicate expressing that the top of the stack is L

$$\exists L\ \underbrace{\mathbf{E}(\ !(\exists f\ call(f) \wedge \mathbf{EX}\ \boxed{L\Gamma^*})}_{\text{L is not a return address of a function call}}\ \mathbf{U}\ (ret \wedge \boxed{L\Gamma^*}))$$

Expressing Obfuscated Returns in SCTPL

l_0: call f
l_1: ...
...
f :...
...
  ret // return

Normal return

Obfuscate
the return

l_0: call f
l_1: ...
...
f : ...
...
  pop eax
  jmp eax

Obfuscated return

$$\exists L\ EF(\overbrace{\exists f\ call(f) \wedge EX\ L\Gamma^*}^{\boxed{\text{L is a return address of a function call}}} \wedge EG!(ret \wedge L\Gamma^*))$$

Expressing Appending Viruses in SCTPL

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

```
L0 : call f
a :
…
f: pop eax
```

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

Malware Detection using SCTPL Model-Checking for PDSs

Binary code $\models$ Malicious behavior ?

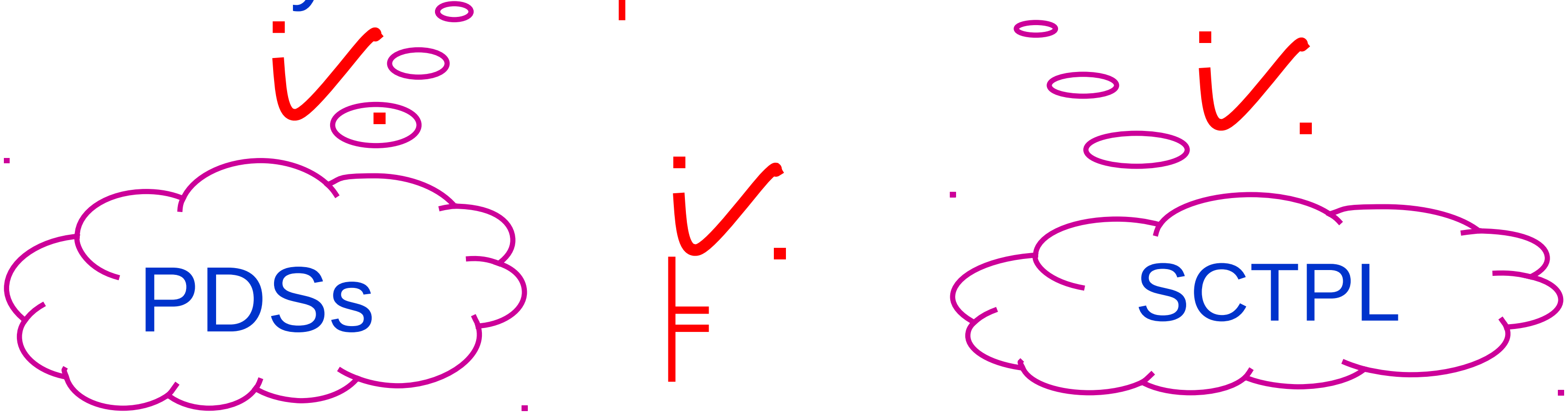
Tool runs out of memory on several malwares

Proposition:

SCTPL is as expressive as CTL with regular valuations (CTLr), but it is exponentially more succinct than CTLr

SCTPL Model-Checking for PDSs

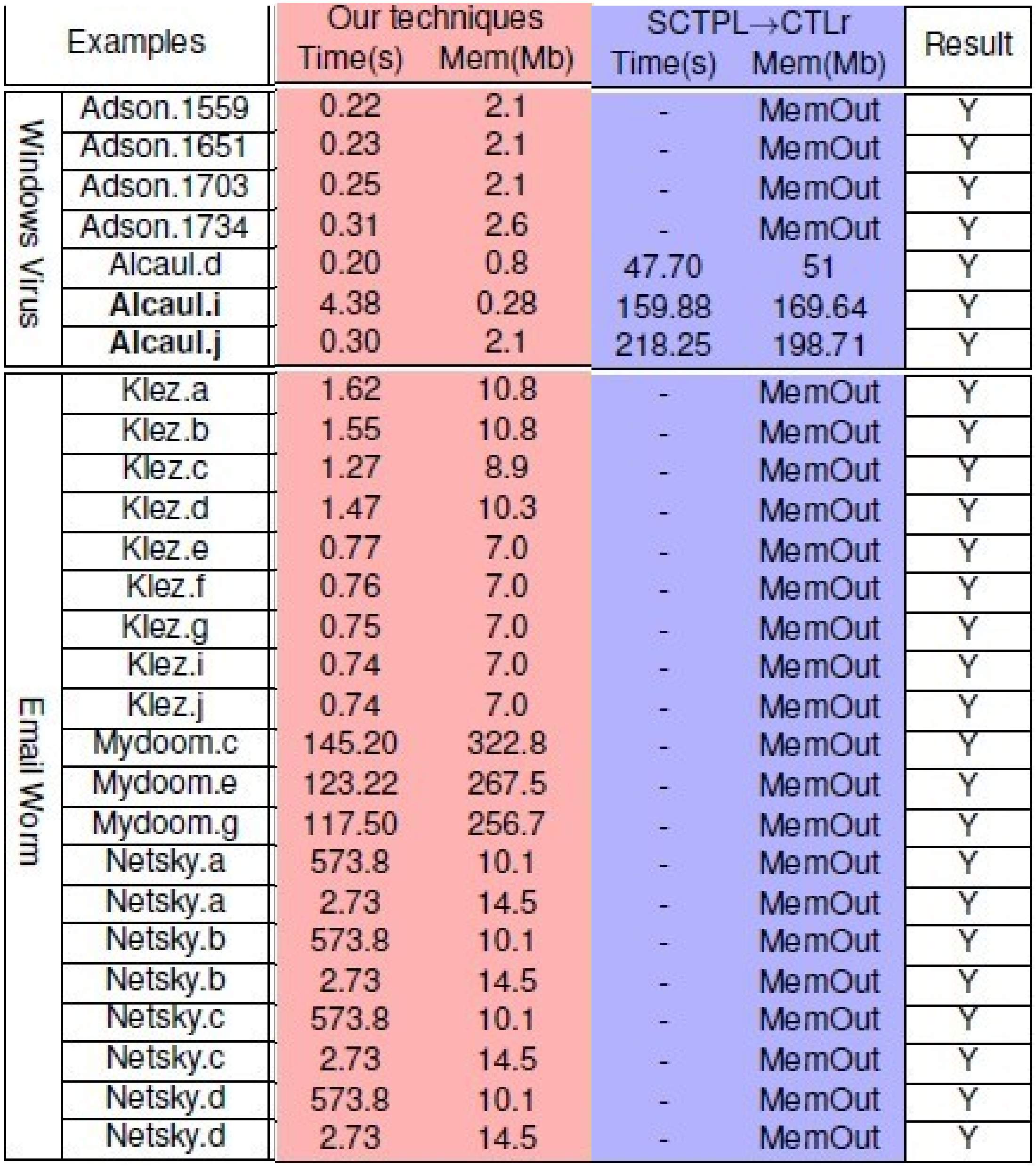
Binary code $\models$ Malicious behavior ?



PDSs $\models$ SCTPL

Thm: Given a PDS *P* and a SCTPL formula ϕ, whether *P* satisfies ϕ can be effectively decided in time O($2^{5(|P|\cdot|\phi|+k)2^{d}}$), where k is the number of states of the finite automata representing regular predicates, d is the number of valuations of variables **Y** over the domain **D.**

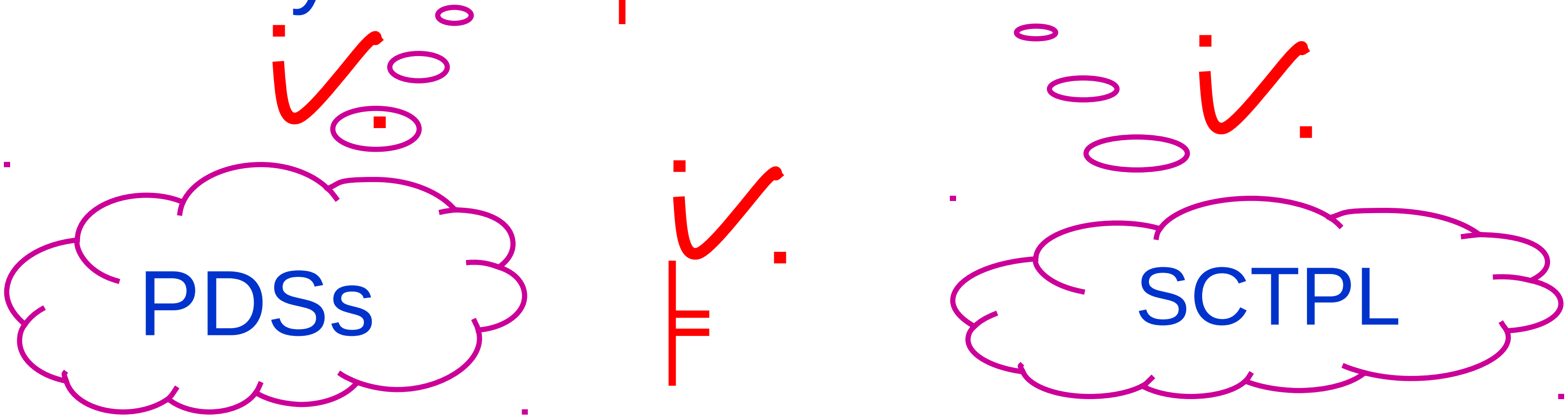
Experiments: SCTPL vs CTLr



Examples		Our techniques		SCTPL→CTLr		Result
		Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	
Windows Virus	Adson.1559	0.22	2.1	-	MemOut	Y
	Adson.1651	0.23	2.1	-	MemOut	Y
	Adson.1703	0.25	2.1	-	MemOut	Y
	Adson.1734	0.31	2.6	-	MemOut	Y
	Alcaul.d	0.20	0.8	47.70	51	Y
	Alcaul.i	4.38	0.28	159.88	169.64	Y
	Alcaul.j	0.30	2.1	218.25	198.71	Y
Email Worm	Klez.a	1.62	10.8	-	MemOut	Y
	Klez.b	1.55	10.8	-	MemOut	Y
	Klez.c	1.27	8.9	-	MemOut	Y
	Klez.d	1.47	10.3	-	MemOut	Y
	Klez.e	0.77	7.0	-	MemOut	Y
	Klez.f	0.76	7.0	-	MemOut	Y
	Klez.g	0.75	7.0	-	MemOut	Y
	Klez.i	0.74	7.0	-	MemOut	Y
	Klez.j	0.74	7.0	-	MemOut	Y
	Mydoom.c	145.20	322.8	-	MemOut	Y
	Mydoom.e	123.22	267.5	-	MemOut	Y
	Mydoom.g	117.50	256.7	-	MemOut	Y
	Netsky.a	573.8	10.1	-	MemOut	Y
	Netsky.a	2.73	14.5	-	MemOut	Y
	Netsky.b	573.8	10.1	-	MemOut	Y
	Netsky.b	2.73	14.5	-	MemOut	Y
	Netsky.c	573.8	10.1	-	MemOut	Y
	Netsky.c	2.73	14.5	-	MemOut	Y
	Netsky.d	573.8	10.1	-	MemOut	Y
	Netsky.d	2.73	14.5	-	MemOut	Y

Malware Detection using SCTPL Satisfiability for PDSs

Binary code ⊨ Malicious behavior ?



How to Make Malware Detection More Efficient

Idea: **reduce the size of program model**

Approach: abstraction

- removes irrelevant instructions from the program
- preserves its malicious behaviors

Collapsing Abstraction

Remove instructions:

- not ed in SCTPL f ula
- don't c the st
- don't chan

Keep tructions:

- us SCTPL formula
- jnz, etc

This abstraction does not preserve all SCTPL formulas

n_1: mov

n_2: dec eax

n_3: push eax

n_4: call GetModu andleA

acle

eax=1

sh eax ax=0

ll GetModule ndleA

Keep origi registers' values

Abstraction

n_3: push eax eax=0

n_4: call GetModuleHandleA

Sublogic SCTPL\X

$\phi ::= b(x_1,\ldots,x_m) \mid e \mid \exists x\ \phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid EG\ \phi \mid E[\phi_1 U \phi_2] \mid call(func) \wedge AX\ e$

Next time operator **AX** is used only to specify the return addresses of the callers.

Formulas of the form "call(func) ^ AX e" are needed to express some malicious behavior, e.g., obfuscated call

$\exists L\ (\ \mathbf{E}\ \ !(\exists f\ call(f) \wedge \mathbf{AX}\ L\boldsymbol{\Gamma^*})\ \mathbf{U}\ (ret \wedge L\boldsymbol{\Gamma^*}\ ))$

Sublogic SCTPL\X

$$\phi ::= b(x_1,\ldots,x_m) \mid e \mid \exists x\ \phi \mid \neg\phi$$
$$\mid \phi_1 \wedge \phi_2 \mid EG\ \phi \mid E[\phi_1 U \phi_2]$$
$$\mid call(func) \wedge AX\ e$$

Next time operator **AX** is used only to specify the return addresses of the callers.

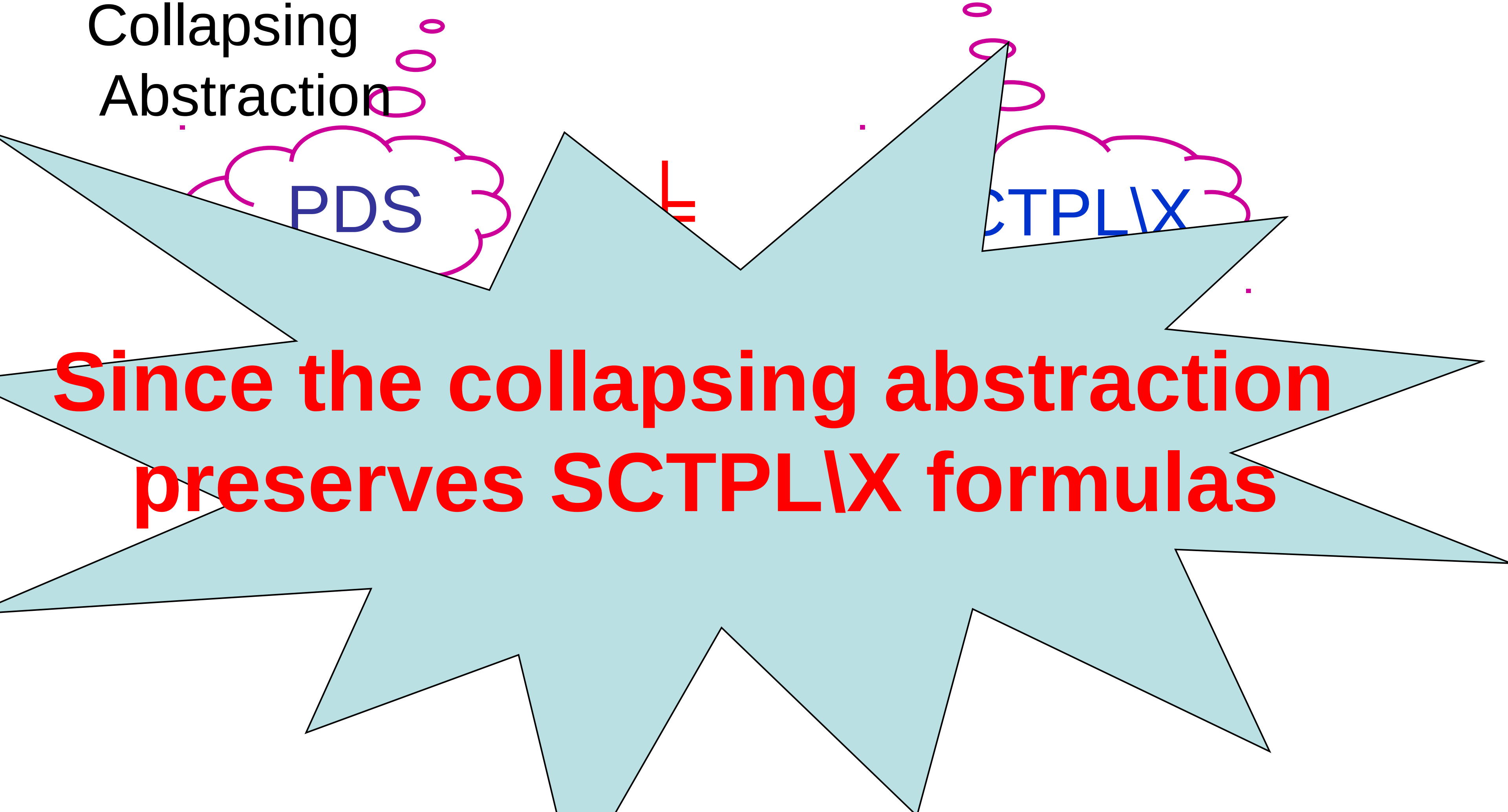
Theorem: A PDS *P* modeling a binary program satisfies a SCTPL\X formula ϕ iff the PDS *P'* modeling the abstracted program satisfies ϕ

SCTPL\X is sufficient to specify malware

- SCTPL formulas using AX or EX other than in the form of call(func) ^ AX e are not robust
- Indeed, suppose a control point n satisfies AXϕ or EXϕ, virus writers can insert any instructions at n without changing the behavior
- This makes specifications using subformulas of the form AXϕ or EXϕ easy to break by virus writers
- Thus, it is recommended to use AF or EF for malware specification instead of AX or EX

Summary of the Approach

Binary code ⊨ Malicious behavior ?



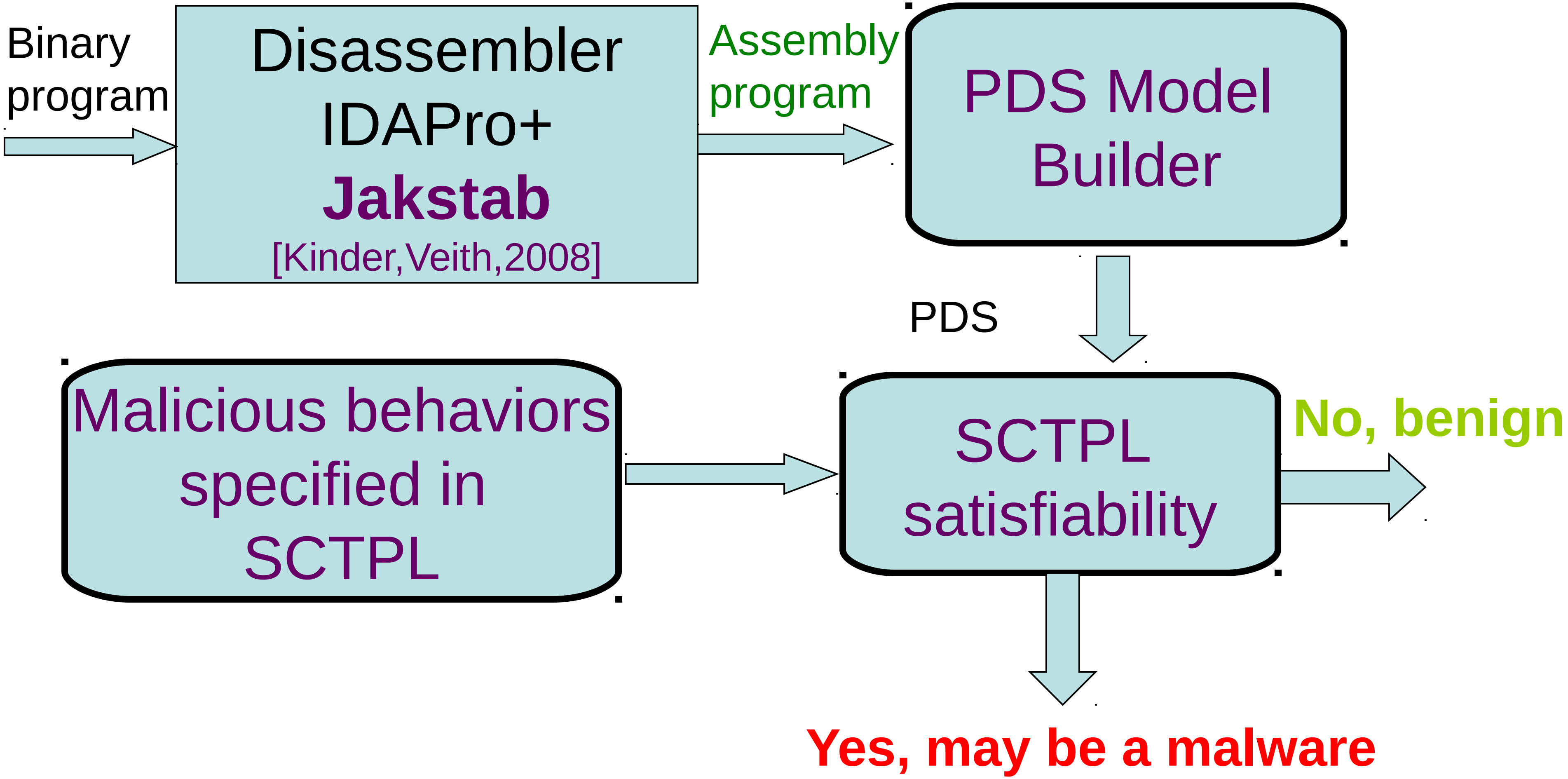
Since the collapsing abstraction preserves SCTPL\X formulas

Implementation

We implemented our techniques in a tool for malware detection

We use Jakstab and IDA Pro to implement the oracle that computes the values of the registers at each control point

The PoMMaDe tool for Malware Detection



Binary program

Disassembler IDAPro+ **Jakstab** [Kinder,Veith,2008]

Assembly program

PDS Model Builder

PDS

Malicious behaviors specified in SCTPL

SCTPL satisfiability

No, benign

Yes, may be a malware

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	POMMADE	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.

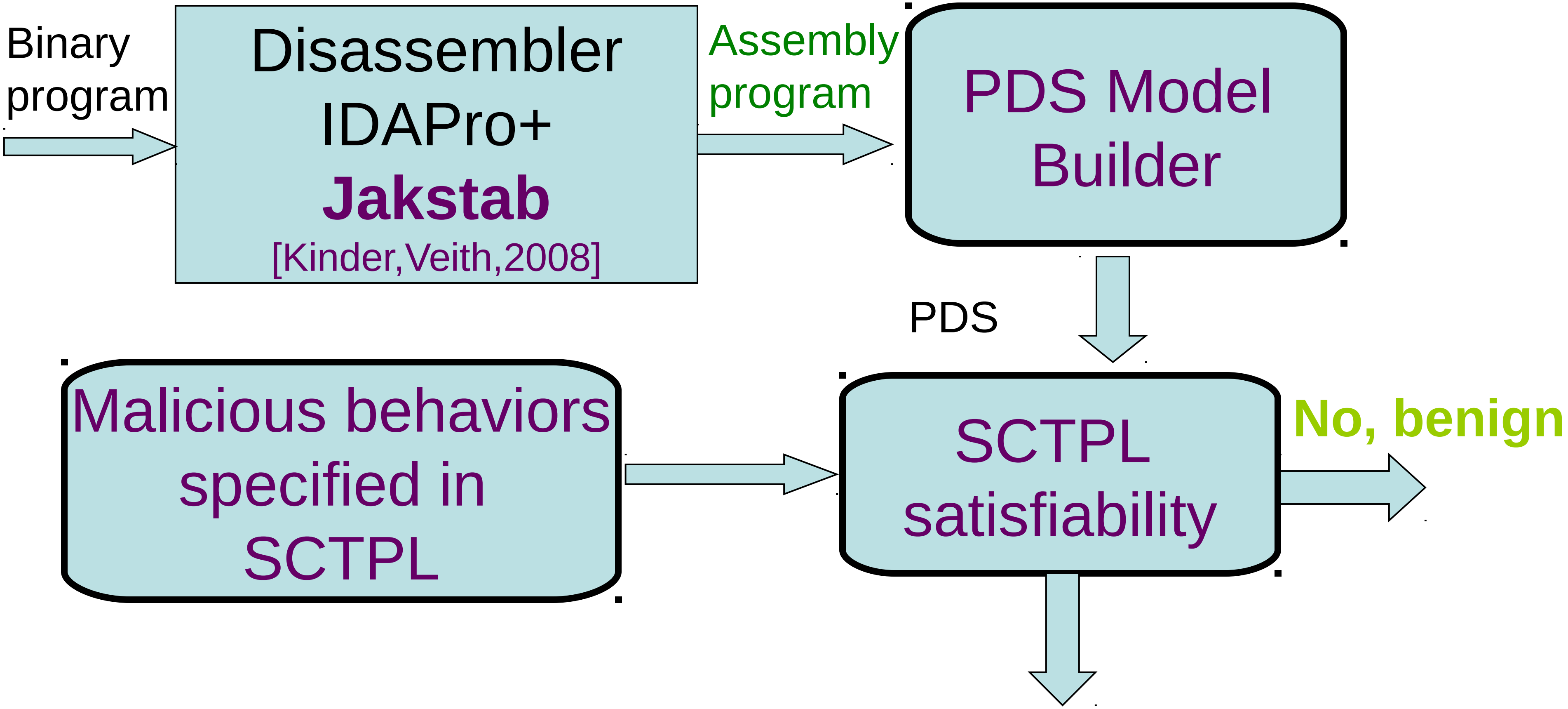
It can

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

Our tool can detect this malware Flame

The PoMMaDe tool for binary code analysis



Binary program

Disassembler
IDAPro+
Jakstab
[Kinder,Veith,2008]

Assembly program

PDS Model Builder

PDS

Malicious behaviors specified in SCTPL

SCTPL satisfiability

No, benign

Yes, may be a malware

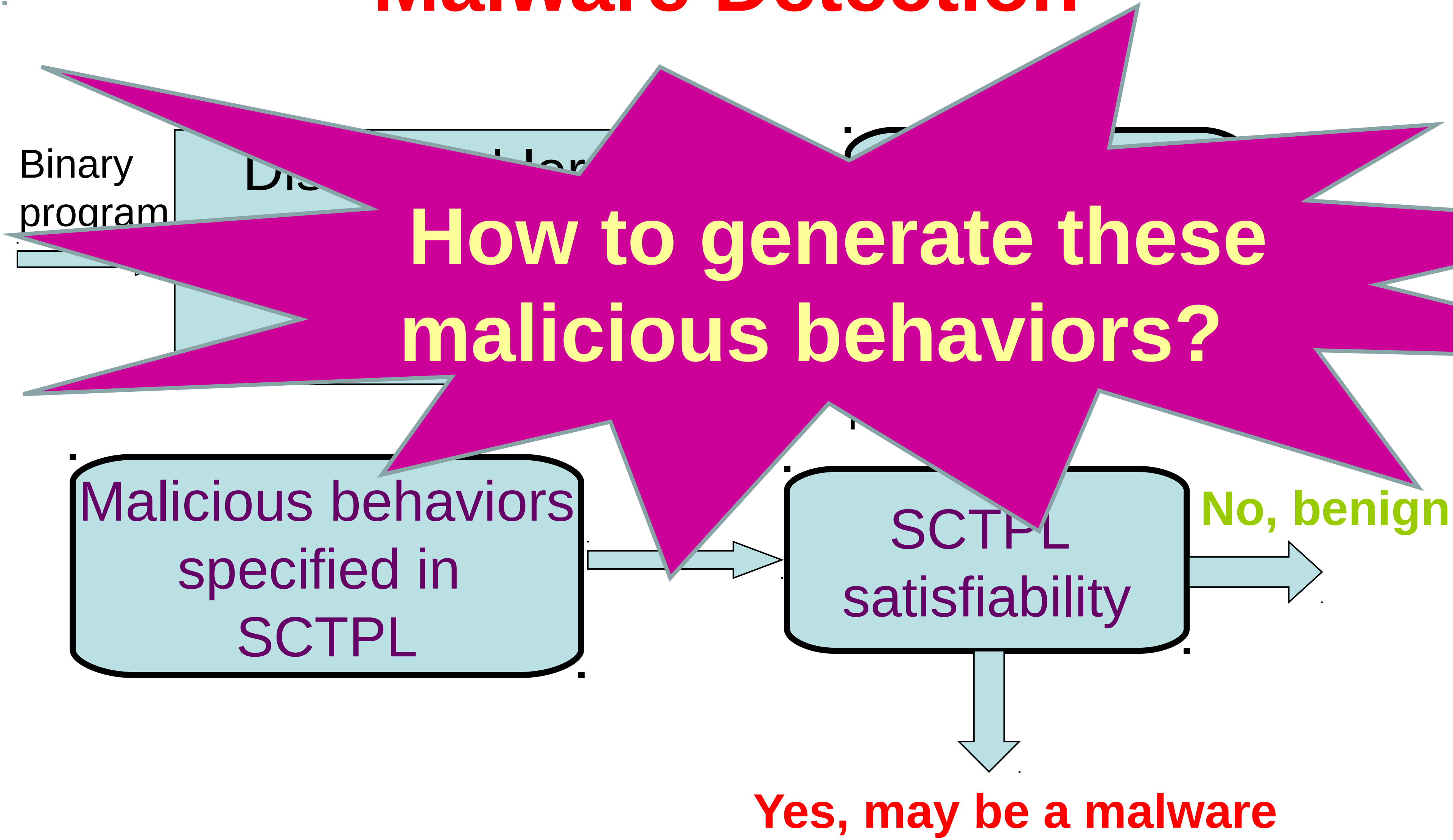
Another application:
Binary code analysis

- Most program analysers operate on source code

- Binary code analysis is needed if source code is not available

- Compilers may introduce errors

The PoMMaDe tool for Malware Detection



Binary program

How to generate these malicious behaviors?

Malicious behaviors specified in SCTPL

SCTPL satisfiability

No, benign

Yes, may be a malware

Malicious Behavior Extraction

- Extracting malicious behaviors requires a huge amount of engineering effort.
  - a tedious and manual study of the code.
  - a huge time for that study.

The main challenge is **how to make this step *automatically*.**

Our goal is …

To extract ***automatically***
the malicious behaviors!

Model Malicious Behaviors

How ?

What is a good model for a malicious behavior??

Trojan Downloader

Transfer data from Internet into a file stored in the system folder, then execute this file.

```
n15     push    0FEh
n16     push    offset dword_4097A4
n17     call    GetSystemDirectoryA
n18     push    0
n19     push    0
n20     lea     eax, [ebp-1Ch]
n21     mov     ebx, eax
n22     push    ebx
n23     push    eax
n24     push    0
n25     call    URLDownloadToFileA
n26     push    5
n27     call    sub_4038B4
n28     push    ebx
n29     call    WinExec
```

*This code is extracted from Trojan-Downloader.Win32.Delf.abk

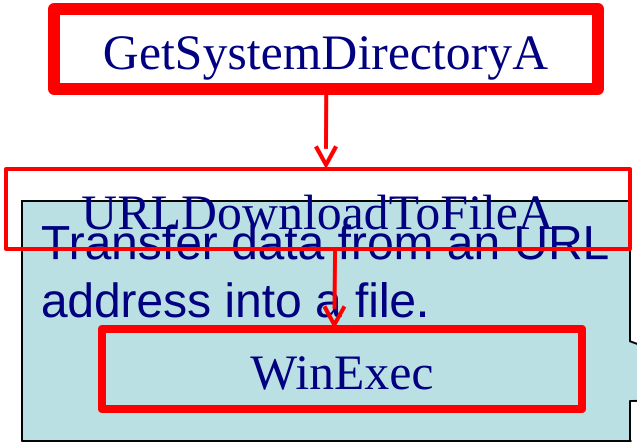
Trojan Downloader

How to extract such graph automatically!!!

Get the … folder.

Transfer data from an URL address into a file.

Executing this file in the system folder.



Malicious API graph

```
        push    0FEh
n16     push    offset dword_4097A4
n17     call    GetSystemDirectoryA
n18     push    0
n19     push    0
n20     lea     eax, [ebp-1Ch]
n21     mov     ebx, eax
n22     push    ebx
n23     push    eax
n24     push    0
n25     call    URLDownloadToFileA
n26     push    5
n27     call    sub_4038B4
n28     push    ebx
n29     call    WinExec
```

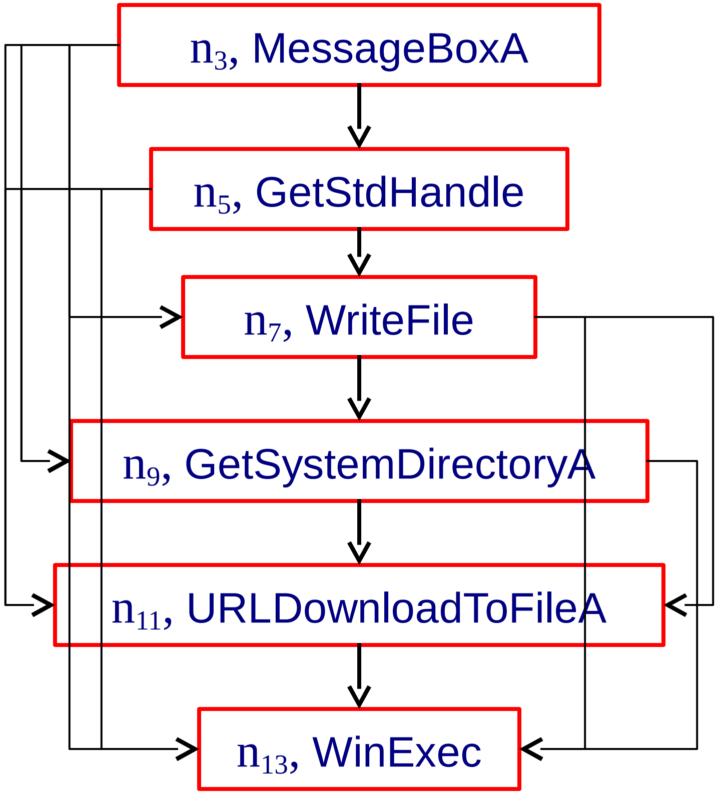
Modeling a program

```
…
n1          push    offset Text
n2          push    0
```

An API call graph represents the order of execution of the different API functions in a program.

```
n10         push    0
n11         call   URLDownloadToFileA
…
n12         push    ebx
n13         call    WinExec
```

*An assembly code of Trojan-Downloader.Win32.Delf.abk



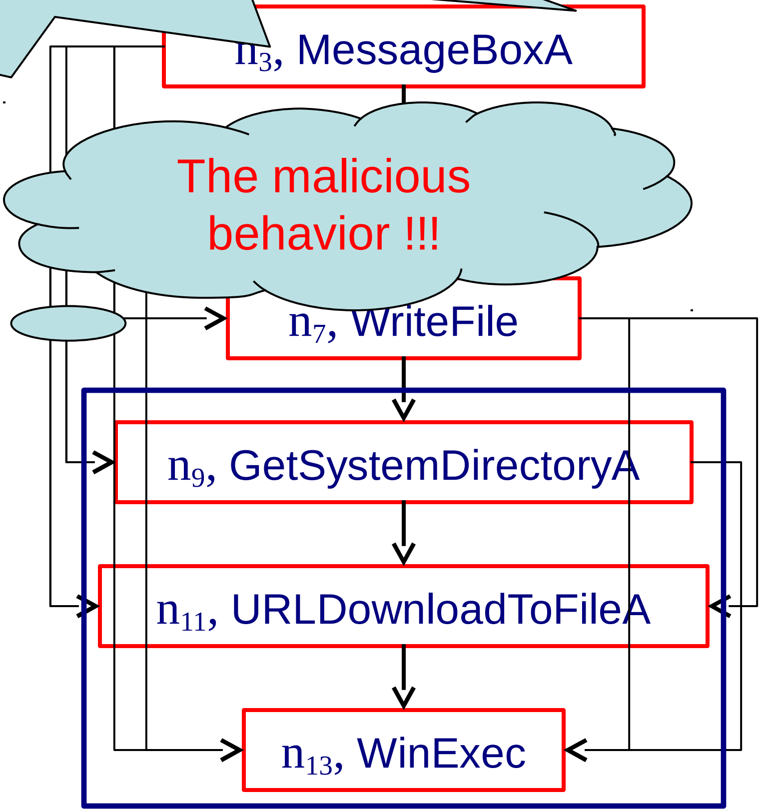
The API call graph

Modeling a program

Our goal is to extract such malicious behavior from this graph.

```
…
n1        push   c    ext
n2        push   0
n3        call   MessageBoxA
…
n4        push   0FFFFFFF5h
n5        call   GetStdHandle
n6        push   eax
n7        call   WriteFile
…
n8        push   offset dword_4097A4
n9        call   GetSystemDirectoryA
…
n10       push   0
n11       call   URLDownloadToFileA
…
n12       push   ebx
n13       call   WinExec
```

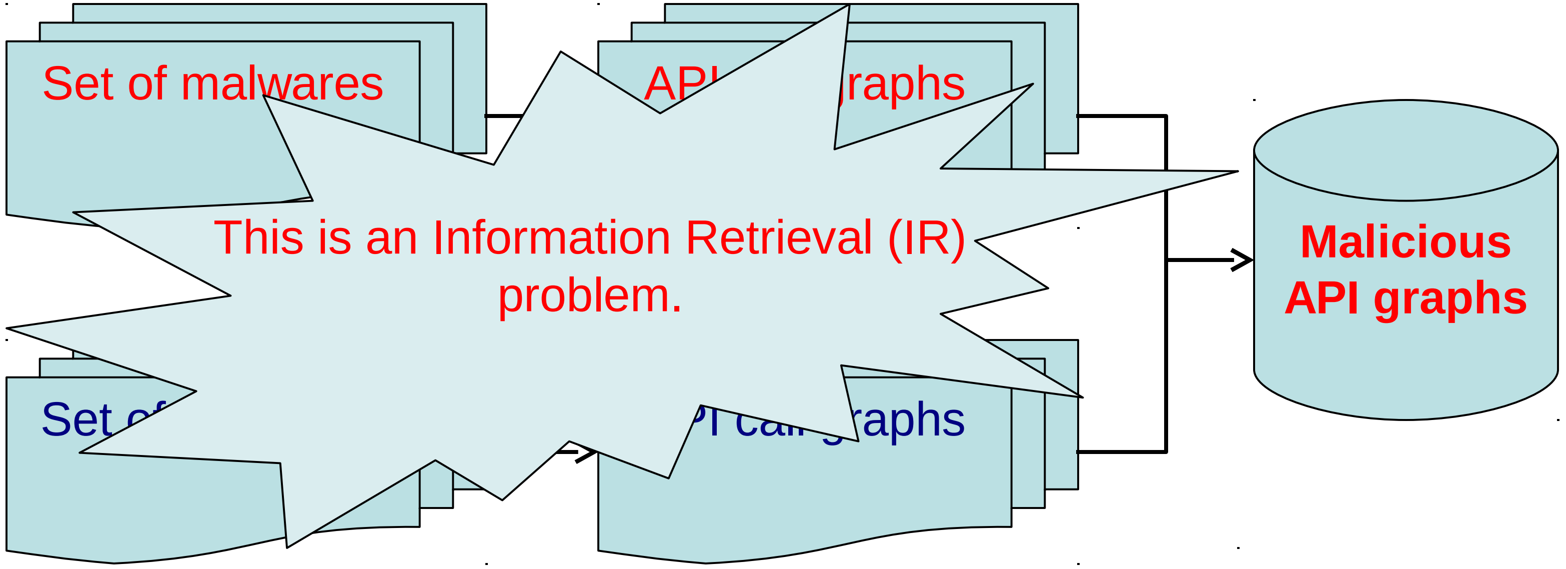
*An assembly code of Trojan-Downloader.Win32.Delf.abk



The malicious behavior !!!

The API call graph

How to extract malicious behaviors?



This is an Information Retrieval (IR) problem.

Our goal:

Isolate the few relevant subgraphs (in malwares) from the nonrelevant ones (in benwares).

IR Problem vs. Our Problem

IR Problem

Retrieve relevant documents and reject nonrelevant ones in a collection of documents.

Our Problem

Isolate the few relevant subgraphs (in malwares) from the nonrelevant ones (in benwares).

Information Retrieval Community

- Extensively studied the problem over the past 35 years.
- Several efficient techniques. Web search, email search, etc.

Our goal is …

Adapt and apply this knowledge and experience of the IR community to our malicious behavior extraction problem.

Information Retrieval

- Information retrieval research has focused on the retrieval of text documents and images.
  - based on extracting from each document a set of terms that allow to distinguish this document from the other documents in the collection.
  - measure the relevance of a term in a document by a term weight scheme.

Term weight scheme in IR

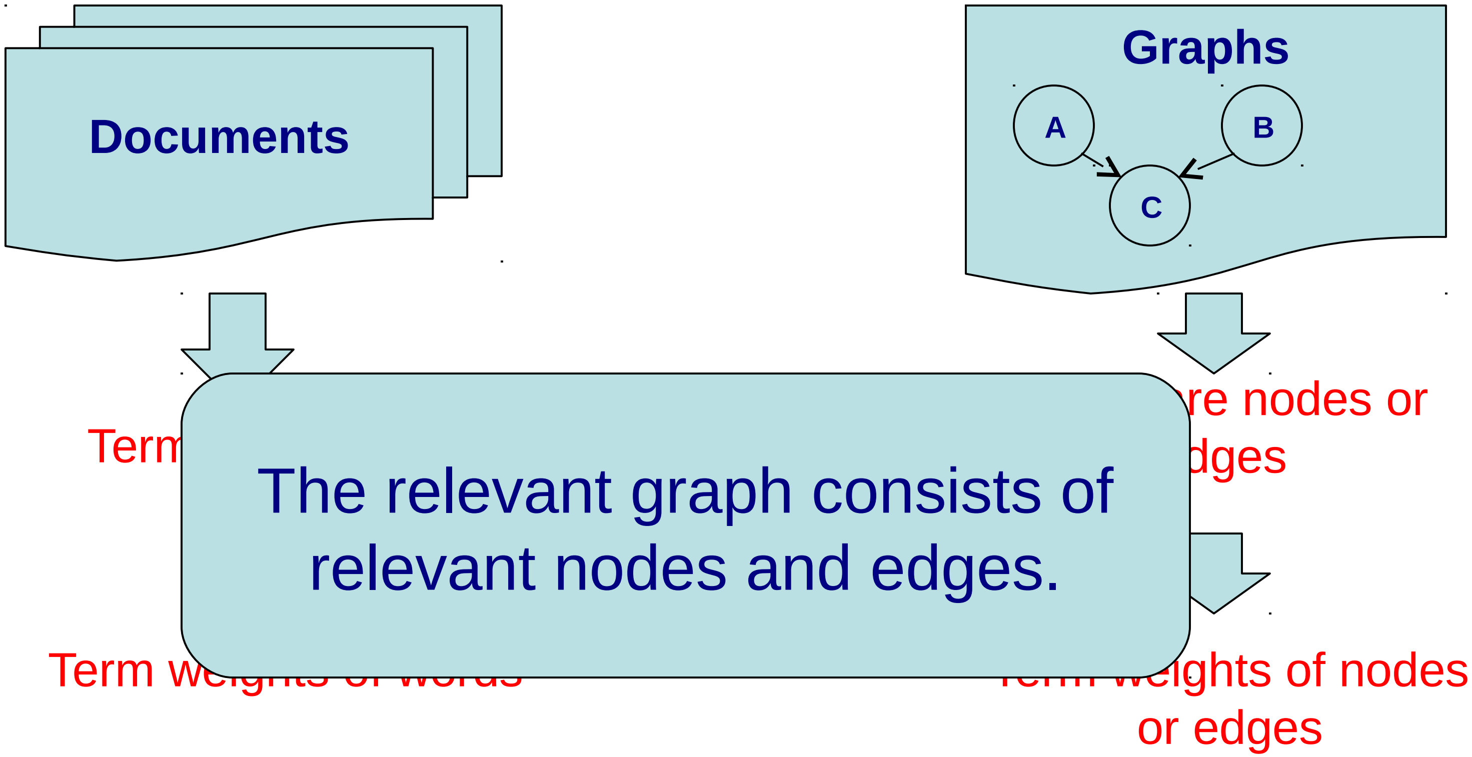
- The term weight represents the relevance of a term in a document.
  - The higher the term weight is, the more relevant the term is in the document.

- A large number of weighting functions have been investigated.
  - The TFIDF scheme is the most popular term weighting in the IR community.

Basic TFIDF scheme

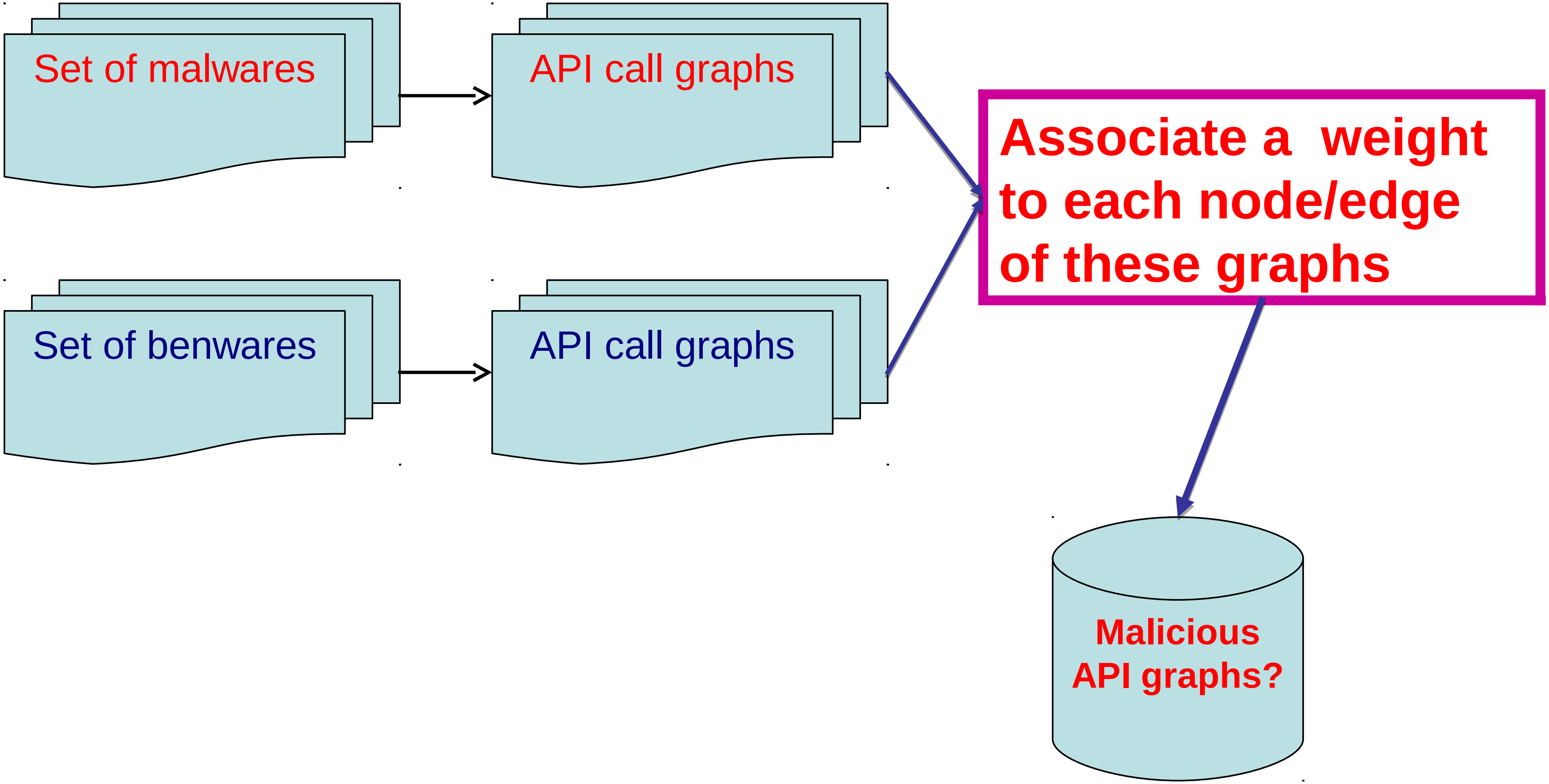
- The TFIDF term weight is measured from the occurrences of terms in a document and their appearances in other documents.

How to apply to our graphs ?



The relevant graph consists of relevant nodes and edges.

Malicious API graph extraction ?



Set of malwares

API call graphs

Set of benwares

API call graphs

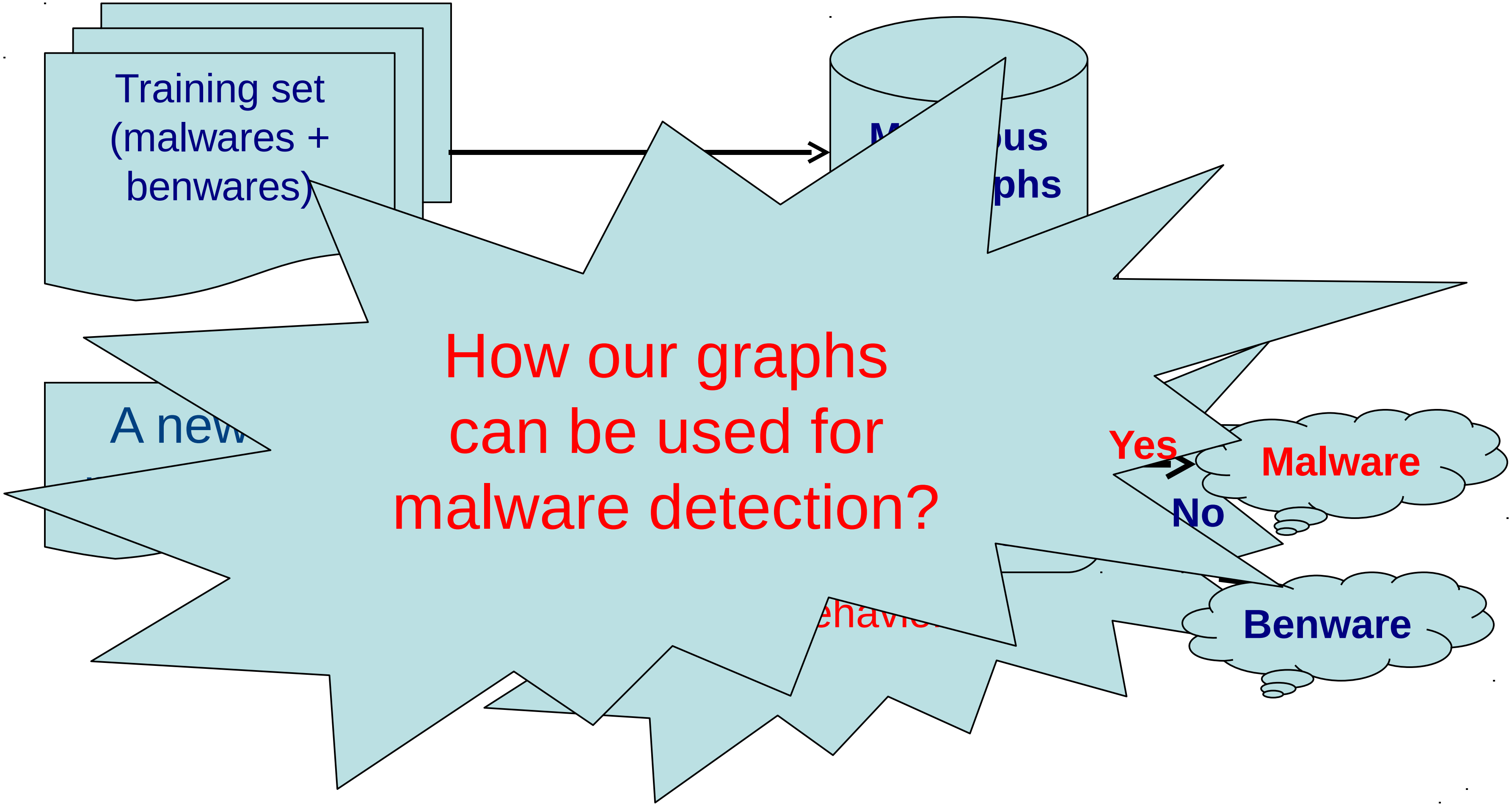
Associate a weight to each node/edge of these graphs

Malicious API graphs?

Construct malicious API graphs

- A malicious API graph consists of nodes and edges with the highest weight.
- Take nodes with highest weight and link them using edges with heighest weight

How to detect malwares?



How our graphs can be used for malware detection?

Experiments

- Apply on a dataset of 1980 benign programs and 3980 malwares collected from Vx Heaven.
  - Training set consists of 1000 benwares and 2420 malwares → extract malicious graphs.
  - Test set consists of 980 benwares and 1560 malwares → for evaluating malicious graphs.

Performance Measurement

- High recall means that most of the relevant items were computed.

$$\mathbf{Recall} = \frac{\mathbf{True\ Positives}}{\boldsymbol{Number\ of\ graphs}}$$

(Detection rate)

99.04%

- High precision means that the technique computes more relevant items than irrelevant.

98.16%

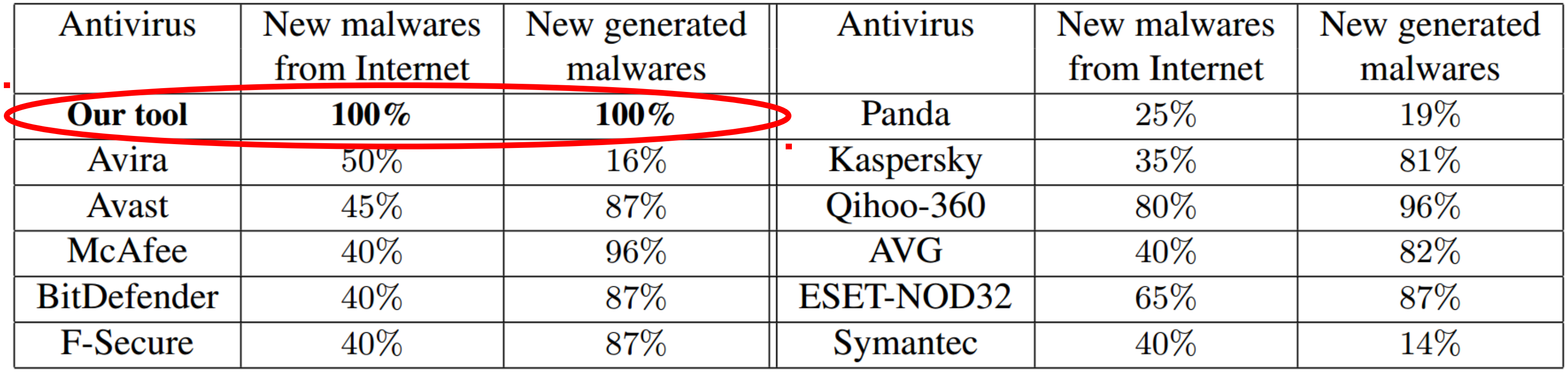
$$\mathbf{Precision} = \frac{\mathbf{True\ Positives}}{\boldsymbol{True\ Postives + False\ Positives}}$$

Comparison with well-known antiviruses

- Detect new unknown malwares
  - 180 new malwares generated by NGVCK, RCWG and VCL32 which are the best known virus generators.
  - 32 new malwares from Internet*.

* https://malwr.com/

Comparison with well-known antiviruses



Antivirus	New malwares from Internet	New generated malwares	Antivirus	New malwares from Internet	New generated malwares
Our tool	**100%**	**100%**	Panda	25%	19%
Avira	50%	16%	Kaspersky	35%	81%
Avast	45%	87%	Qihoo-360	80%	96%
McAfee	40%	96%	AVG	40%	82%
BitDefender	40%	87%	ESET-NOD32	65%	87%
F-Secure	40%	87%	Symantec	40%	14%

A comparison of our method against well-known antiviruses.

The problem is …

- Extracting malicious behaviors requires a huge amount of engineering effort.
  - a tedious and manual study of the code.
  - a huge time for that study.

The main challenge is **to avoid this manual work.**

What about machine learning?

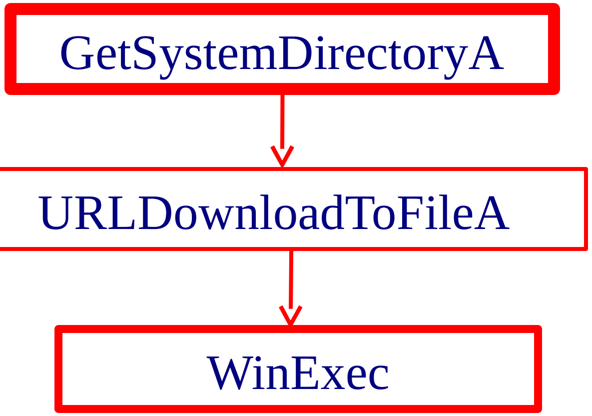
Apply machine learning to detect malwares without extracting the malicious behaviors.

Our goal is…

To implement machine learning for malware detection.

Model Malicious Behaviors

Trojan Downloader



GetSystemDirectoryA

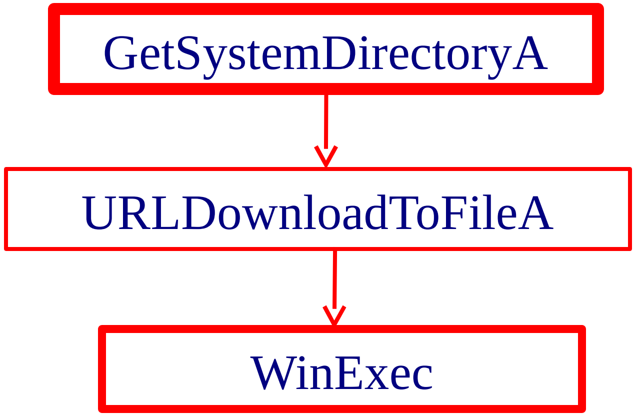
URLDownloadToFileA

WinExec

Malicious API graph

```
n15    push   0FEh
n16    push   offset dword_4097A4
n17    call   GetSystemDirectoryA
n18    push   0
n19    push   0
n20    lea    eax, [ebp-1Ch]
n21    mov    ebx, eax
n22    push   ebx
n23    push   eax
n24    push   0
n25    call   URLDownloadToFileA
n26    push   5
n27    call   sub_4038B4
n28    push   ebx
n29    call   WinExec
```

Trojan Downloader


GetSystemDirectoryA

URLDownloadToFileA

WinExec

Malicious API graph

n_{15} push 0FEh

n_{16} push offset dword_4097A4

n_{17} call GetSystemDirectoryA

n_{18} push 0

How can we model a program to learn such a graph?

n_{24} push 0

n_{25} call URLDownloadToFileA

n_{26} push 5

n_{27} call sub_4038B4

n_{28} push ebx

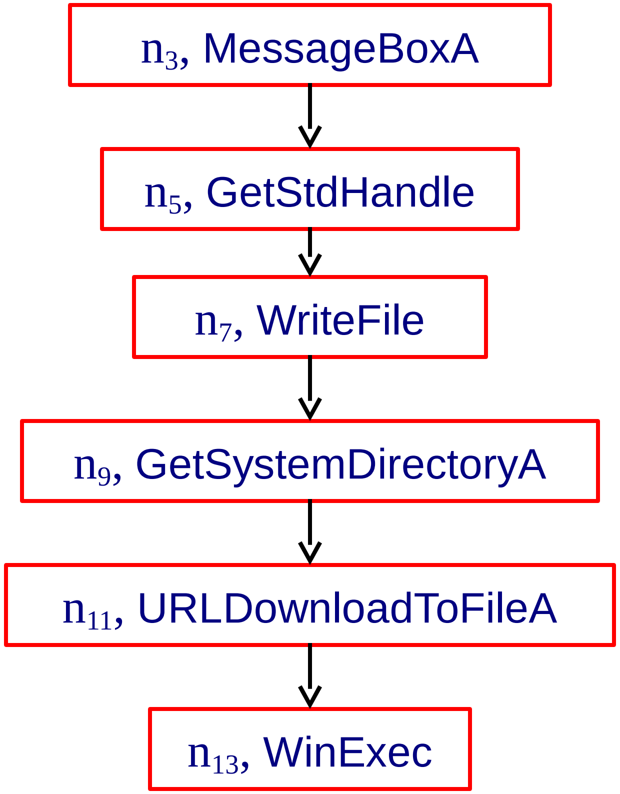
n_{29} call WinExec

Modeling a program

```
…
n1      push   offset Text
…
n10     push   0
n11     call   URLDownloadToFileA
…
n12     push   ebx
n13     call   WinExec
```

An API call graph represents the order of execution of the different API functions in a program.

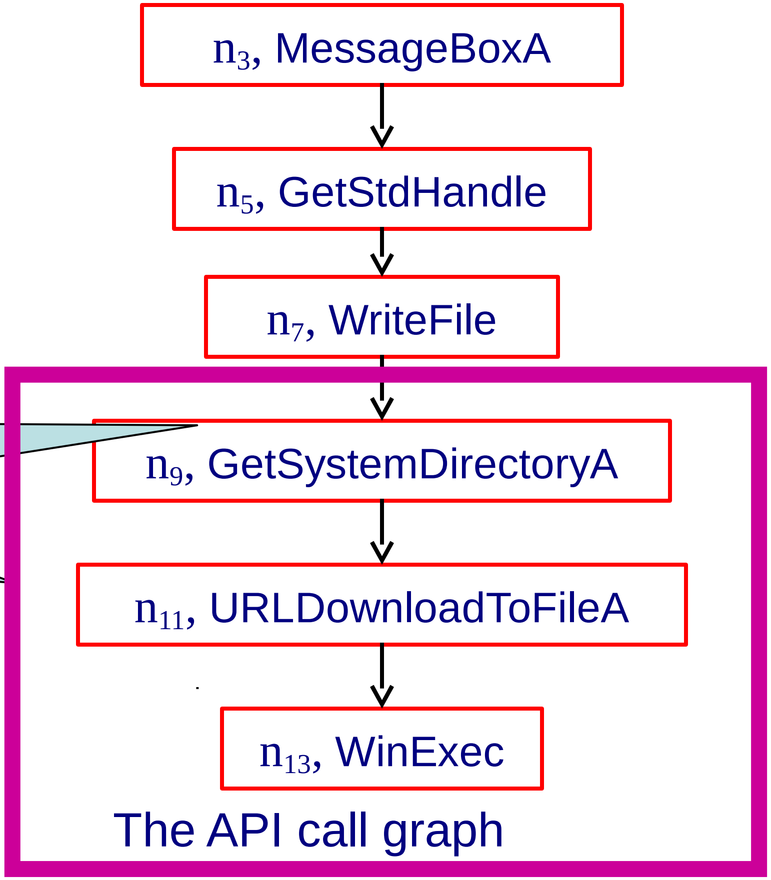
*An assembly code of Trojan-Downloader.Win32.Delf.abk



The API call graph

Modeling a program

How to learn this behavior?



n_3, MessageBoxA

n_5, GetStdHandle

n_7, WriteFile

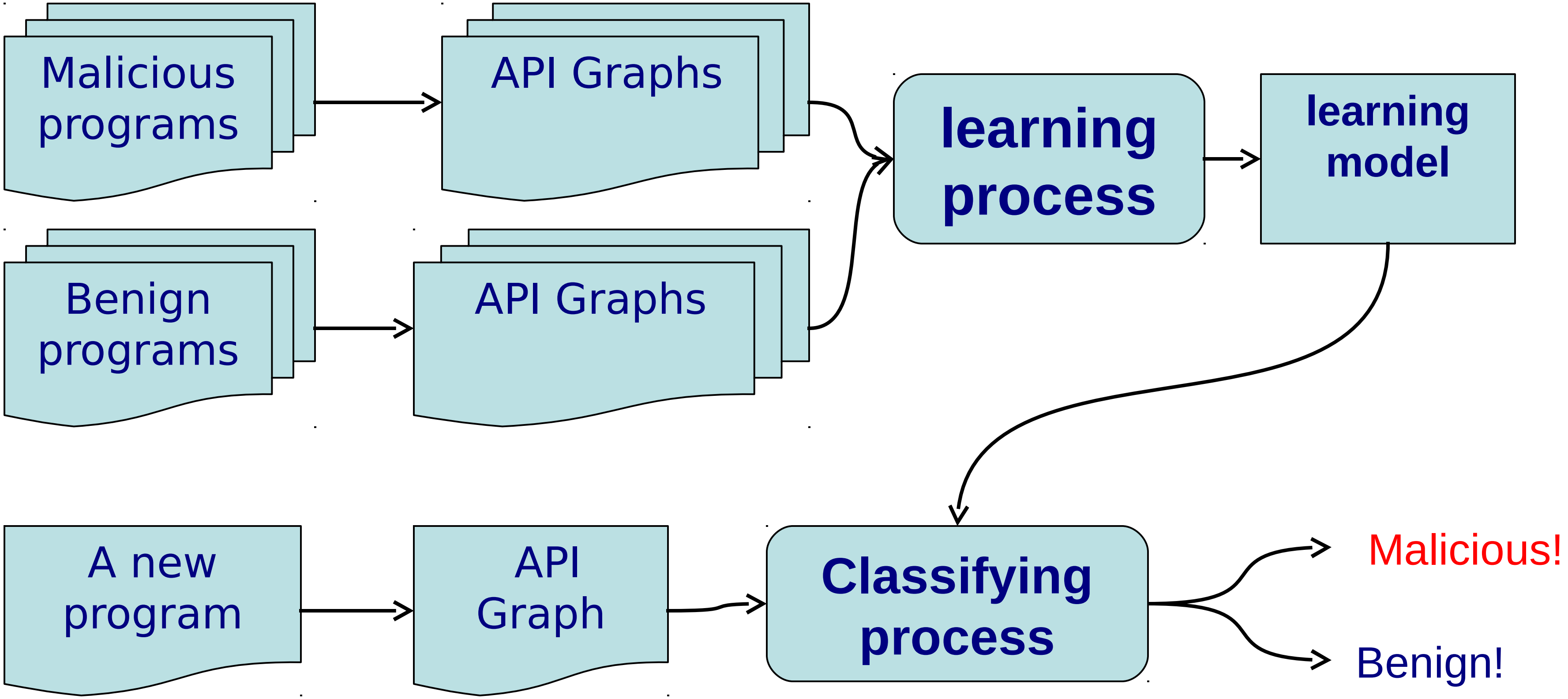
n_9, GetSystemDirectoryA

n_{11}, URLDownloadToFileA

n_{13}, WinExec

The API call graph

Our approach



Malicious programs

API Graphs

Benign programs

API Graphs

learning process

learning model

A new program

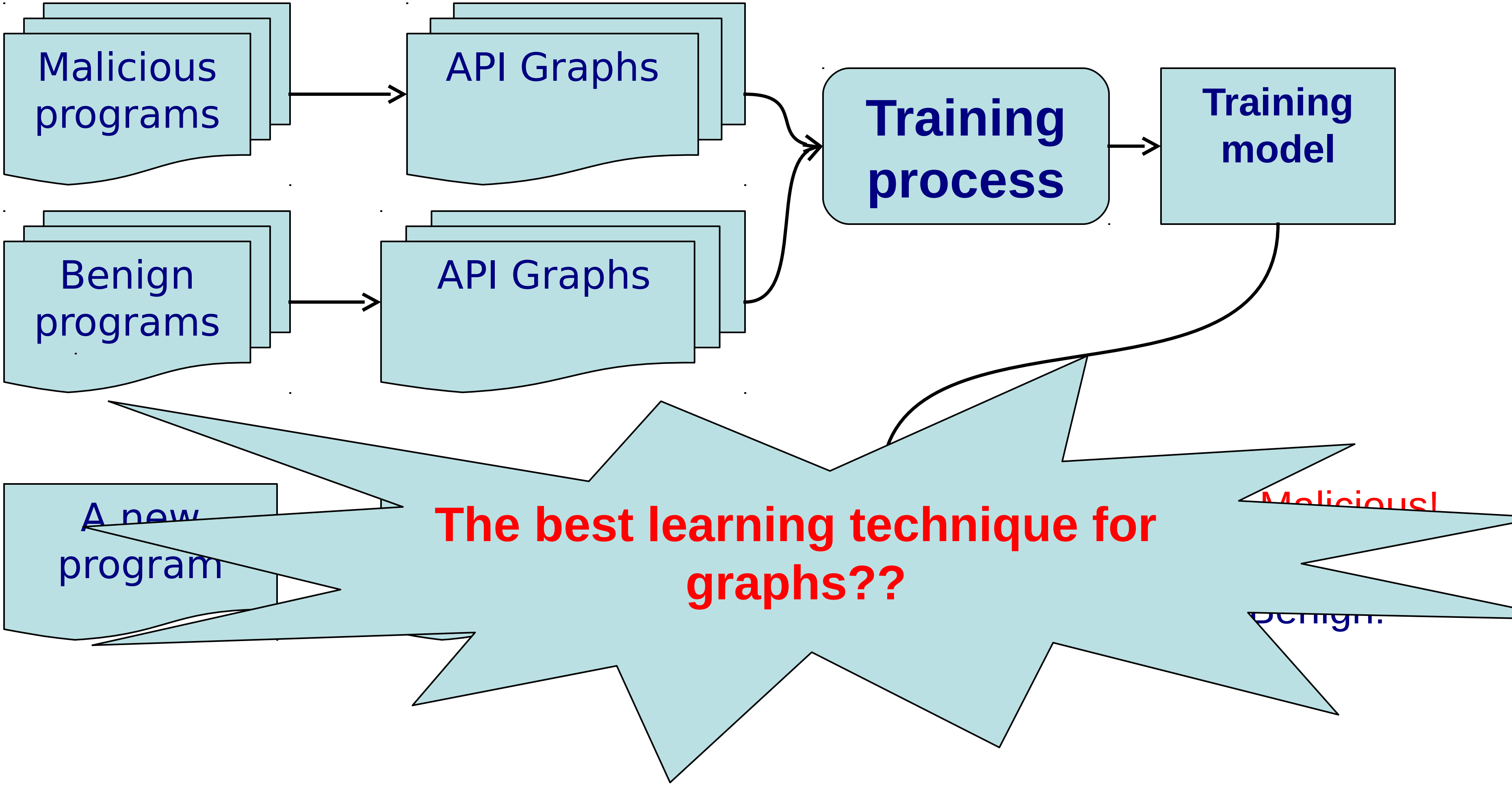
API Graph

Classifying process

Malicious!

Benign!

Our approach



Malicious programs

API Graphs

Benign programs

API Graphs

Training process

Training model

A new program

Malicious!

Benign!

The best learning technique for graphs??

The problem…

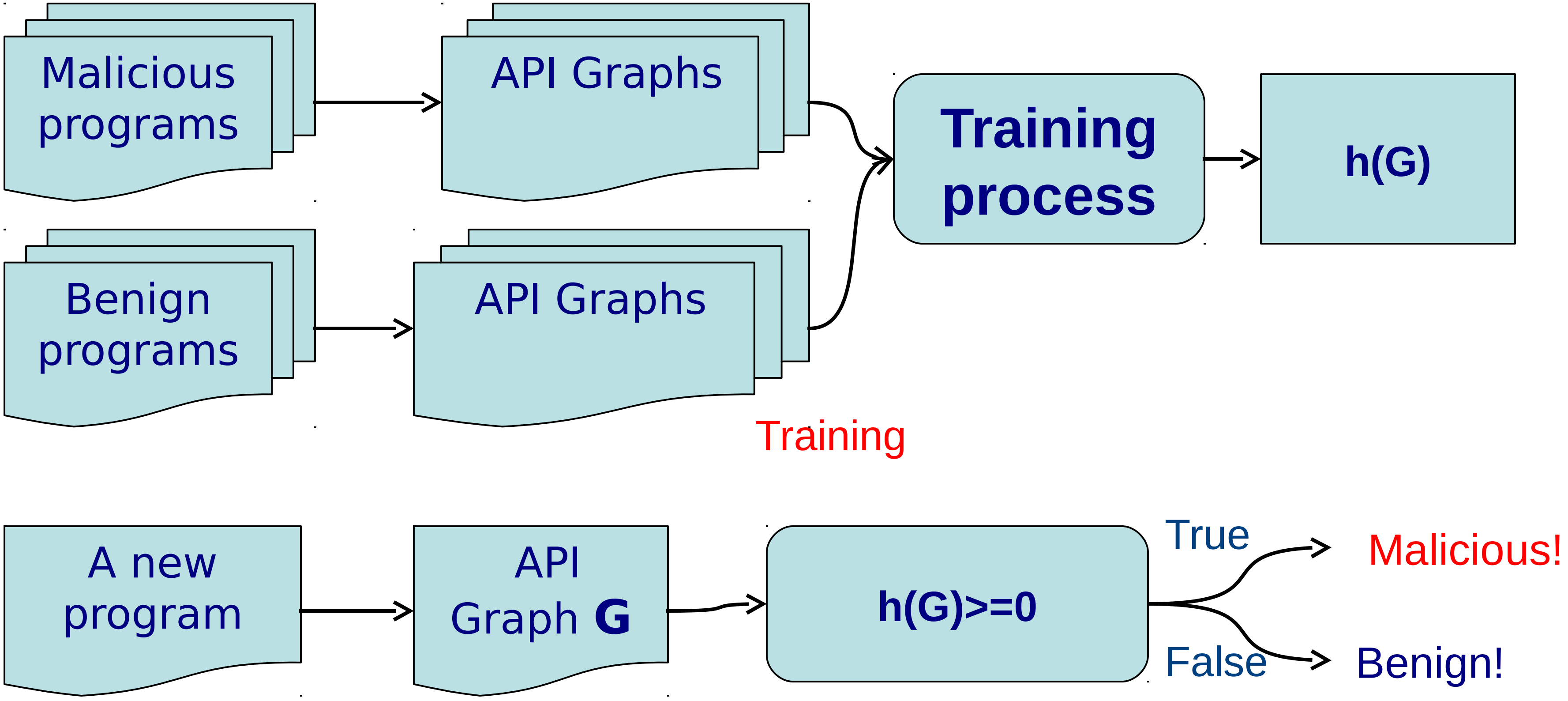
- The existing machine learning techniques can mainly be applied to vectorial data.
- But our data are API call graphs.
  - Not vectorial data!!!

We need to use a learning technique for graphs.

Kernel based SVM

- The best learning technique that can be applied for graphs
  - Kernel based Support Vector Machines.

Summary of our approach



Malicious programs

API Graphs

Benign programs

API Graphs

Training process

h(G)

Training

A new program

API Graph **G**

h(G)>=0

True

Malicious!

False

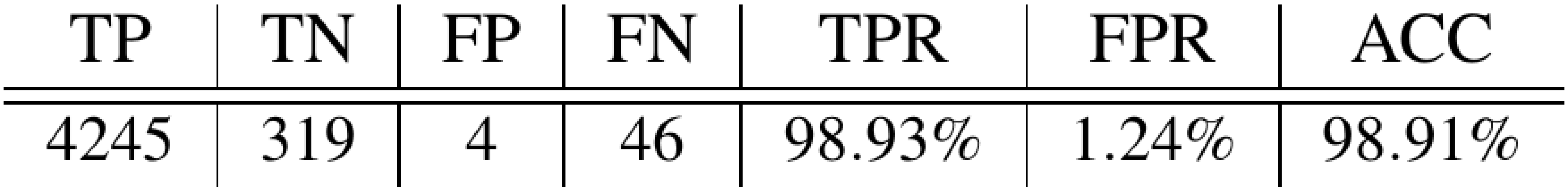
Benign!

Detecting

Experiments

- We evaluate this technique on the dataset of 2323 benign programs and 6291 malicious programs.
  - Training set of 2000 malwares and 2000 benwares.
  - Test set of 4291 malwares and 323 benwares.

The results on the dataset



TP	TN	FP	FN	TPR	FPR	ACC
4245	319	4	46	98.93%	1.24%	98.91%

TP: True Positives

TN: True Negatives

FP: False Positives

FN: False Negatives

TPR: True Positive Rates

TPR = TP/(TP+FN)

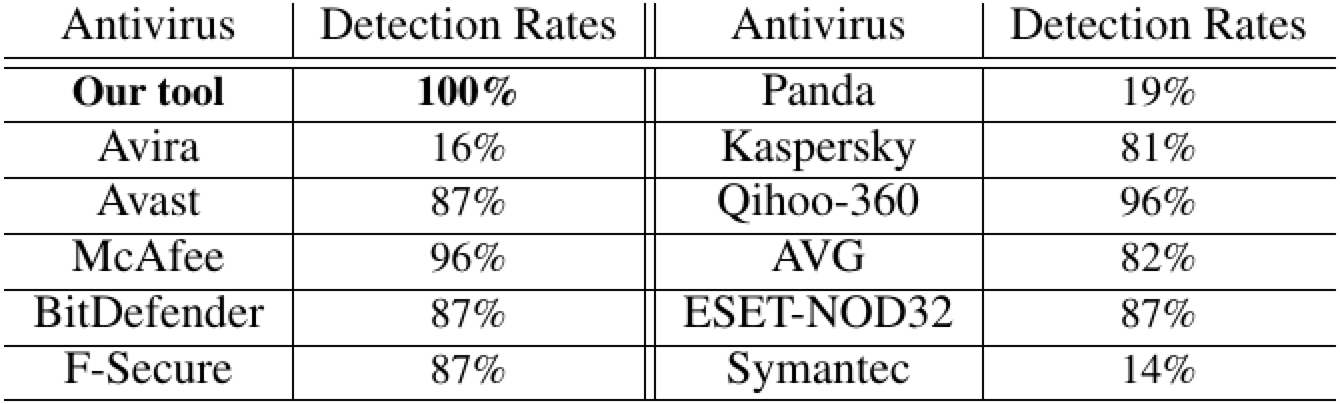
FPR: False Positive Rates

FPR = FP/(TN+FP)

ACC = (TP+TN)/(TP+FN+TN+FP): Accuracy

Anti-virus software comparison

- We generate 180 malwares from virus generators (RCWG, VCL32 and NGVCK).



Antivirus	Detection Rates	Antivirus	Detection Rates
Our tool	**100%**	Panda	19%
Avira	16%	Kaspersky	81%
Avast	87%	Qihoo-360	96%
McAfee	96%	AVG	82%
BitDefender	87%	ESET-NOD32	87%
F-Secure	87%	Symantec	14%

Behavior Signatures

- SCTPL or malicious API graphs to represent malicious behaviors
- These correspond to **behavior signatures**

Questions?