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 in 2010 is more than **1.5 billion**.
- The worm MyDoom slowed 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.

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 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**: Executed binary code in a virtual environment.
  - Checks program’s behavior only in a limited time interval

**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.

\[
\begin{align*}
L_0 &: \text{call } f \\
L_1 &: \ldots \\
& \quad \ldots \\
& \quad \ldots \\
& \quad f : \text{function } f
\end{align*}
\]

\[
\begin{align*}
L_0 &: \text{push } L_1 \\
L'0 &: \text{jmp } f \\
L_1 &: \ldots \\
& \quad \ldots \\
& \quad \ldots \\
& \quad f : \text{function } f
\end{align*}
\]
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 system.
- Virus writers try to hide these calls.

Important to analyse the program's stack

Solution:
Use pushdown systems to model programs
Pushdown Systems

PDS = finite automaton + Stack

\[ P = (P, \Gamma, \Delta), \]

- \( P \) is a finite set of control states
- \( \Gamma \) is the stack alphabet
- \( \Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*) \) is a finite set of transitions
- A configuration is a pair \( <p, \omega> \in P \times \Gamma^* \)
- If \( <p, \alpha> \rightarrow <p', \omega> \in \Delta \), then, for every \( u \in \Gamma^* \),
  \[ <p, \alpha u> \Rightarrow <p', \omega u> \]
From Binary Codes to PDSs
Difficulty:

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

\begin{align*}
I_1 &: \text{mov eax, 1} \\
I_2 &: \text{dec eax} \\
I_3 &: \text{push eax} \\
I_4 &: \text{call GetModuleHandleA} \\
I_5 &: \ldots
\end{align*}

$g_0 =$ entry point of GetModuleHandleA

Control states of PDS = control points of program
Stack alphabet = return addresses + registers’ values

$\begin{align*}
I_1 \rightarrow I_2 \rightarrow I_3 \\
\text{Push 0} \rightarrow \text{Push } I_5
\end{align*}$
Malicious behaviors?

Binary code \quad \Rightarrow \quad \text{Malicious behavior?}

- PDS

- Specification formalism?
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.

How to describe this specification?

```
mov eax, 0
push eax
call GetModuleHandleA
```
Specification of malicious behaviors?
Example: fragment of email worm Avron

In CTL (Branching-time temporal logic):
\[
\text{mov}(eax,0) \land \text{EX} (\text{push}(eax) \land \text{EX} \ \text{call GetModuleHandleA})
\]

**EX p:** there is a path where \( p \) holds at the next state
Specification of malicious behaviors?
Example: fragment of email worm Avron

In CTL (Branching-time temporal logic):
\[
\text{mov}(\text{eax}, 0) \land \text{EX} (\text{push}(\text{eax}) \land \text{EX call GetModuleHandleA}) \\
\lor \\
\text{mov}(\text{ebx}, 0) \land \text{EX} (\text{push}(\text{ebx}) \land \text{EX call GetModuleHandleA}) \\
\lor \\
\text{mov}(\text{ecx}, 0) \land \text{EX} (\text{push}(\text{ecx}) \land \text{EX call GetModuleHandleA}) \\
\lor \\
\ldots \text{ all the other registers}
\]

\text{EX} p: there is a path where \( p \) holds at the next state
Specification of malicious behaviors?
Example: fragment of email worm Avron

In CTL (Branching-time temporal logic):
\[
\text{mov(eax,0)} \mathop{\land} \text{EX (push(eax)} \mathop{\land} \text{EX call GetModuleHandleA)} \\
\text{mov(ebx,0)} \mathop{\land} \text{EX (push(ebx)} \mathop{\land} \text{EX call GetModuleHandleA)} \\
\text{mov(ecx,0)} \mathop{\land} \text{EX (push(ecx)} \mathop{\land} \text{EX call GetModuleHandleA)} \\
\text{...... all the other registers}
\]

\text{EX p:} \text{ there is a path where } p \text{ holds at the next state}
Specification of malicious behaviors?

Example: fragment of email worm Avron

CTPL = CTL + variables + \exists, \forall

In CTL:

\text{mov(eax,0)^EX (push(eax)^EX call GetModuleHandleA)}

\text{mov(ebx,0)^EX (push(ebx)^EX call GetModuleHandleA)}

\text{mov(ecx,0)^EX (push(ecx)^EX call GetModuleHandleA)}

\ldots all the other registers

In CTPL:

\exists r (\text{mov(r,0)^EX (push(r)^EX call GetModuleHandleA)})
Specification of malicious behaviors?

Example: fragment of email worm Avron

CTPL = CTL + variables + \exists

CTPL cannot describe the stack: needed for malicious behaviors description

In CTPL:

\exists r (\text{mov}(r,0)^{\text{EX}} (\text{push}(r)^{\text{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.

In CTPL:

\[ \exists r \left( \text{mov}(r,0)^{\text{EX}} \left( \text{push}(r)^{\text{EX}} \text{ call } \text{GetModuleHandleA} \right) \right) \]
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.

In CTPL:
\[ \exists r \left( \text{mov}(r, 0) \land \text{EX}(\text{push}(r) \land \text{EX} \text{call GetModuleHandleA}) \right) \]

Our solution: Consider predicates over the stack

In SCTPL:
\[ \text{EF} \left( \text{call GetModuleHandleA} \land 0 \right) \]

**EF p:** there is a path where \( p \) holds in the future
\[ \phi ::= b \ | \neg \phi \ | \phi \land \phi \ | \text{EX } \phi \ | \text{E}[\phi \ U \ \phi] \ | \text{EG } \phi \]
SCTPL Logic

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

- \( y \in Y \), a set of variables over a finite domain \( D \)
SCTPL Logic

\[ \phi ::= b(y_1, \ldots, y_n) | \neg \phi | \phi \land \phi | \text{EX} \phi | \text{E}[\phi U \phi] | \text{EG} \phi | \exists y \phi \]

- \( y \in Y \), a set of variables over a finite domain \( D \)
SCTPL Logic

\( \phi ::= b(y_1, \ldots, y_n) \mid \neg \phi \mid \phi \land \phi \mid \text{EX} \ \phi \mid \text{E} [\phi \cup \phi] \mid \text{EG} \ \phi \mid \exists y \ \phi \mid e \)

- \( y \in Y \), a set of variables over a finite domain \( D \)
- \( e \) is a regular expression over \( Y \cup \Gamma \)
Expressing Obfuscated Calls in SCTPL

Normal function call

Obfuscate the call

L is not a return address of a function call

\[ \exists L \ E ( ! ( \exists f \ call(f) \land EX L \Gamma^*) \lor (\text{ret} \land L \Gamma^*) ) \]
Expressing Obfuscated Returns in SCTPL

\[ l_0: \text{call } f \]
\[ l_1: ... \]
\[ ... \]
\[ f: ... \]
\[ ... \]
\[ \text{ret // return} \]

Normal return

- Obfuscate the return

\[ l_0: \text{call } f \]
\[ l_1: ... \]
\[ ... \]
\[ f: ... \]
\[ ... \]
\[ \text{pop eax} \]
\[ \text{jmp eax} \]

Obfuscated return

\[ \exists L \ EF(\exists f \text{ call}(f) \land EX LG^* \land EG!(ret \land LG^*)) \]

L is a return address of a function call.
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

\[ \forall f \forall a \left( \left( \text{call}(f) \land AX \ a \uparrow^* \right) \implies AF \neg r \left( \text{pop}(r) \land a \uparrow^* \right) \right) \]

- \( a \) is a return address of a procedure call.
Proposition:
SCTPL is as expressive as CTL with regular valuations (CTLr), but it is exponentially more succinct than CTLr.
Thm: Given a PDS $P$ and a SCTPL formula $\phi$, whether $P$ satisfies $\phi$ 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

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

Binary code \models \text{Malicious behavior} ?

PDSs \models \text{SCTPL}
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 used in SCTPL formula
• don’t change the stack
• don’t change the control flow

Keep instructions:
• used in SCTPL formula
• push, pop
• call, ret, jmp, jz, jnz, etc

This abstraction does not preserve all SCTPL formulas

Keep original registers’ values

Abstraction

Oracle

\[ n_1: \text{mov} \quad \text{eax}, 1 \]

\[ n_2: \text{dec} \quad \text{eax} \]

\[ n_3: \text{push} \quad \text{eax} \]

\[ n_4: \text{call} \ \text{GetModuleHandleA} \]
Sublogic SCTPL\X

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

Next time operator \( \text{AX} \) is used only to specify the return addresses of the callers.

Formulas of the form “\text{call(func)} \land \text{AX} e” are needed to express some malicious behavior, e.g., obfuscated call

\[ \exists L ( E \neg (\exists f \text{call(f)} \land \text{AX} L \Gamma^*) U (\text{ret} \land L \Gamma^*) ) \]
Sublogic SCTPL\X

\[ \phi ::= b(x_1,\ldots,x_m) \mid e \mid \exists x \ \phi \mid \neg \phi \]
\[ |\phi_1 \land \phi_2 |\text{EG } \phi | E[\phi_1 U \phi_2] \]
\[ | \text{call(func)} \uparrow \text{AX } e \]

Next time operator \textbf{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 \( \phi \) iff the PDS \( P' \) modeling the abstracted program satisfies \( \phi \).
SCTPL\X is sufficient to specify malware

- SCTPL formulas using $AX$ or $EX$ other than in the form of $\text{call(func)} \land AX e$ are not robust.
- Indeed, suppose a control point $n$ satisfies $AX\phi$ or $EX\phi$, virus writers can insert any instructions at $n$ without changing the behavior.
- This makes specifications using subformulas of the form $AX\phi$ or $EX\phi$ 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 $\models$ Malicious behavior? 

Collapsing Abstraction

PDS $\models$ SCTPL\X 

Since the collapsing abstraction preserves SCTPL\X formulas
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

- **Disassembler**: IDAPro+ Jakstab [Kinder, Veith, 2008]
- **Assembly program**
- **Binary program**

**PDS Model Builder**

**SCTPL satisfiability**

- **Malicious behaviors specified in SCTPL**

Yes, may be a malware

No, benign
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

<table>
<thead>
<tr>
<th>Generator</th>
<th>No. Of Variants</th>
<th>NGVCK</th>
<th>VCL32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avira</td>
<td>23%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Kaspersky</td>
<td>18%</td>
<td>68%</td>
<td>0%</td>
</tr>
<tr>
<td>Avast</td>
<td>68%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Qihoo 360</td>
<td>100%</td>
<td>11%</td>
<td>97%</td>
</tr>
<tr>
<td>McAfee</td>
<td>100%</td>
<td>100%</td>
<td>81%</td>
</tr>
<tr>
<td>AVG</td>
<td>100%</td>
<td>97%</td>
<td>0%</td>
</tr>
<tr>
<td>BitDefender</td>
<td>100%</td>
<td>100%</td>
<td>76%</td>
</tr>
<tr>
<td>Eset Nod32</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>F-Secure</td>
<td>99%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Norton</td>
<td>2%</td>
<td>0%</td>
<td>46%</td>
</tr>
<tr>
<td>Panda</td>
<td>18%</td>
<td>0%</td>
<td>30%</td>
</tr>
<tr>
<td>Trend Micro</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
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

Disassembler
IDAPro+
Jakstab
[Kinder, Veith, 2008]

Assembly program

PDS Model Builder

PDS

SCTPL satisfiability

Malicious behaviors specified in SCTPL

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

Disassembled Binary Program

How to generate these malicious behaviors?

Malicious behaviors specified in SCTPL

SCTPL satisfiability

Yes, may be a malware

No, benign
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.

push 0FEh
push offset dword_4097A4
call GetSystemDirectoryA
push 0
push 0
lea eax, [ebp-1Ch]
mov ebx, eax
push ebx
push eax
push 0
call URLDownloadToFileA
push 5
call sub_4038B4
push ebx
call WinExec

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

Get the path of the system folder.

Transfer data from an URL address into a file.

Executing this file in the system folder.

GetSystemDirectoryA

URLDownloadToFileA

WinExec

Malicious API graph

How to extract such graph automatically!!!
Modeling a program

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

The API call graph

- \( n_1 \): push offset Text
- \( n_2 \): push 0
- \( n_3 \): call MessageBoxA
- \( n_5 \): GetStdHandle
- \( n_7 \): WriteFile
- \( n_9 \): GetSystemDirectoryA
- \( n_{11} \): URLDownloadToFileA
- \( n_{13} \): WinExec
Modeling a program

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

The API call graph

- n3, MessageBoxA
- n7, WriteFile
- n9, GetSystemDirectoryA
- n11, URLDownloadToFileA
- n13, WinExec

The malicious behavior !!!

Program assembly code of Trojan-Downloader.Win32.Delf.abk
How to extract malicious behaviors?

Set of malwares
API call graphs
Malicious API graphs

This is an Information Retrieval (IR) problem.

Set of benwares
API call graphs

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

Our goal is …
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.

- Documents
- Term weights of words
- Graphs
- Term weights of nodes or edges
- Relevant nodes or 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?

Training set (malwares + benwares) → Malicious graphs

How our graphs can be used for malware detection?

A new program API call graph

Yes

Malware

No

Benware

Check common paths
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.
  \[
  \text{Recall} = \frac{\text{True Positives}}{\text{Number of graphs}} \quad (\text{Detection rate})
  \]
  99.04%

• High **precision** means that the technique computes more relevant items than irrelevant.
  \[
  \text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}
  \]
  98.16%
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

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>New malwares from Internet</th>
<th>New generated malwares</th>
<th>Antivirus</th>
<th>New malwares from Internet</th>
<th>New generated malwares</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our tool</td>
<td>100%</td>
<td>100%</td>
<td>Panda</td>
<td>25%</td>
<td>19%</td>
</tr>
<tr>
<td>Avira</td>
<td>50%</td>
<td>16%</td>
<td>Kaspersky</td>
<td>35%</td>
<td>81%</td>
</tr>
<tr>
<td>Avast</td>
<td>45%</td>
<td>87%</td>
<td>Qihoo-360</td>
<td>80%</td>
<td>96%</td>
</tr>
<tr>
<td>McAfee</td>
<td>40%</td>
<td>96%</td>
<td>AVG</td>
<td>40%</td>
<td>82%</td>
</tr>
<tr>
<td>BitDefender</td>
<td>40%</td>
<td>87%</td>
<td>ESET-NOD32</td>
<td>65%</td>
<td>87%</td>
</tr>
<tr>
<td>F-Secure</td>
<td>40%</td>
<td>87%</td>
<td>Symantec</td>
<td>40%</td>
<td>14%</td>
</tr>
</tbody>
</table>

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

Malicious API graph

- **GetSystemDirectoryA**
- **URLDownloadToFileA**
- **WinExec**
Trojan Downloader

Malicious API graph

GetSystemDirectoryA

URLDownloadToFileA

WinExec

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

n₁₅ push 0FEh
n₁₆ push offset dword_4097A4
n₁₇ call GetSystemDirectoryA
n₁₈ push 0
n₁₉ push 0
n₂₀ push ebx
n₂₁ call WinExec
n₂₂ push 5
n₂₃ call sub_4038B4
n₂₄ push ebx
n₂₅ call URLDownloadToFileA
Modeling a program

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

The API call graph:

- $n_3$, MessageBoxA
- $n_5$, GetStdHandle
- $n_7$, WriteFile
- $n_9$, GetSystemDirectoryA
- $n_{11}$, URLDownloadToFileA
- $n_{13}$, WinExec

The API call graph diagram shows the sequence of API calls executed by the program, starting with pushing an offset to the text and ending with calling WinExec.

The diagram includes:

1. $n_1$: push offset Text
2. $n_3$: call MessageBoxA
3. $n_5$: call GetStdHandle
4. $n_7$: call WriteFile
5. $n_9$: call GetSystemDirectoryA
6. $n_{11}$: call URLDownloadToFileA
7. $n_{13}$: call WinExec
Modeling a program

The API call graph

How to learn this behavior?
Our approach

- Malicious programs
  - API Graphs
  - learning process
  - learning model

- Benign programs
  - API Graphs
  - learning process
  - learning model

- A new program
  - API Graph
  - Classifying process
    - Malicious!
    - Benign!
Our approach

Malicious programs ➔ API Graphs ➔ Training process ➔ Training model

Benign programs ➔ API Graphs

A new program

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.
The best learning technique that can be applied for graphs

– Kernel based Support Vector Machines.
Summary of our approach

Malicious programs → API Graphs → Training process → h(G)

Benign programs → API Graphs → Training process → h(G)

A new program → API Graph G → h(G) >= 0 → Malicious! or Benign!

Training

Detecting
Experiments

• We evaluate this technique on the dataset of 2323 benign programs and 6291 malicious programs.
  – Test set of 4291 malwares and 323 benwares.
The results on the dataset

<table>
<thead>
<tr>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>TPR</th>
<th>FPR</th>
<th>ACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4245</td>
<td>319</td>
<td>4</td>
<td>46</td>
<td>98.93%</td>
<td>1.24%</td>
<td>98.91%</td>
</tr>
</tbody>
</table>

TP: True Positives
TN: True Negatives
FP: False Positives
FN: False Negatives
TPR: True Positive Rates
FPR: False Positive Rates

\[
TPR = \frac{TP}{TP+FN}\]

\[
FPR = \frac{FP}{TN+FP}\]

\[
ACC = \frac{TP+TN}{TP+FN+TN+FP}: \text{Accuracy}\]
Anti-virus software comparison

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

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Detection Rates</th>
<th>Antivirus</th>
<th>Detection Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our tool</td>
<td>100%</td>
<td>Panda</td>
<td>19%</td>
</tr>
<tr>
<td>Avira</td>
<td>16%</td>
<td>Kaspersky</td>
<td>81%</td>
</tr>
<tr>
<td>Avast</td>
<td>87%</td>
<td>Qihoo-360</td>
<td>96%</td>
</tr>
<tr>
<td>McAfee</td>
<td>96%</td>
<td>AVG</td>
<td>82%</td>
</tr>
<tr>
<td>BitDefender</td>
<td>87%</td>
<td>ESET-NOD32</td>
<td>87%</td>
</tr>
<tr>
<td>F-Secure</td>
<td>87%</td>
<td>Symantec</td>
<td>14%</td>
</tr>
</tbody>
</table>
Behavior Signatures

• SCTPL or malicious API graphs to represent malicious behaviors
• These correspond to behavior signatures
Questions?