Making sense of a million samples per day

Behavior-based Methods for Automated, Scalable Malware Analysis

Prof. Stefano Zanero, Politecnico di Milano
Therefore I say: One who knows the enemy and knows himself will not be in danger in a hundred battles. One who does not know the enemy but knows himself will sometimes win, sometimes lose. One who does not know the enemy and does not know himself will be in danger in every battle.

[Sun-Tsu, Ping-fa]
Motivation: The evolution of malware

• 1980s-1990s: **self-replicant** code (self-similar, written as test of skill, not primarily harmful)

• 2005-2015: predominance of diffusive, stealth **malware** (aim: monetization) with many variants

• Growth of an underground economy (crime-as-a-service)

• 2010-on: also state-sponsored attacks and surveillance
Stefano Zanero

Motivation: Cyberwar and attribution

- Stuxnet: designed to sabotage Iran's nuclear facilities
- Duqu: discovered a few months later, possibly created earlier, same platform as Stuxnet; uses zero-day; designed to collect data on the Iranian nuclear program (which ended up in the hands of UN)
- Issue of attribution and analysis
Large scale malware analysis: the issue

• Analysts are way too few, code is way too much

• Need better ways to automatically analyze malware
  ▪ What does it do?
  ▪ Where does it come from?
  ▪ What is new about it? Why was it created?

What enables the enlightened rulers and good generals to conquer the enemy at every move and achieve extraordinary success is foreknowledge.

Sun-tzu
Static vs. dynamic approaches

**Static approaches**

+ Complete analysis
- Difficult to extract semantics
- Obfuscation / packing

**Dynamic approaches**

- “Dormant” code
+ Easy to see “behaviors”
+ Malware unpacks itself
Our Approach

• Turn weakness into strength, and strength into weakness: leverage code reuse between malware samples to our advantage
  - Automatically generate semantic-aware models of code implementing a given malicious behavior
  - Use these models to statically detect the malicious functionality in samples that do not perform that behavior during dynamic analysis
  - Use a variation of this technique to study malware evolution over time

Disorder came from order, fear came from courage, weakness came from strength.
Sun-tzu
A. Dynamic Behavior Identification

B. Extracting Genotypes Models

C. Finding Dormant Functionality

Malware sample → Behaviors → Genotype models → New malware samples
Dynamic Behavior Identification

A. Dynamic Behavior Identification

O' -> B_1 -> B_2 -> B_3 -> ... -> B_n

Malware sample → Behaviors

B. Extracting Genotypes Models

γ_1

C. Finding Dormant Functionality

γ_2

γ_3

γ_n

Genotype models

New malware samples
Dynamic Behavior Identification

- Run malware in instrumented sandbox (Anubis: anubis.iseclab.org)

- Dynamically detect a behavior $B$ \textit{(phenotype)}

- Map $B$ to the set $R_B$ of system/API call instances responsible for it

- $R_B$ output of the behavior identification phase
Behavior Detection Examples

- **spam**: send SMTP traffic on port 25
  - network level detection
- **sniff**: open promiscuous mode socket
  - system call level detection
- **rpcbind**: attempt remote exploit against a specific vulnerability
  - network level detection, with snort signature
- **drop**: drop and execute a binary
  - system call level detection, using data flow information
- ...

Extracting Genotype Models

A. Dynamic Behavior Identification

B. Extracting Genotypes Models

C. Finding Dormant Functionality

\[ O' \rightarrow B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \cdots \rightarrow B_n \]

Behaviors

Genotype models

New malware samples

Malware sample
Extracting Genotype Models: Goals

- Identified genotype should be precise and complete
  - **Complete**: include all of the code implementing B
  - **Precise**: do not include code that is not specific to B (utility functions,..)

- We proceed by *slicing* the code, then *filtering* it to remove support code, and *germinating* to complete it
Slicing

- Start from relevant calls $R_B$

- Include into slice $\phi$ instructions involved in:
  - preparing input for calls in $R_B$
    - follow data flow dependencies backwards from call inputs
  - processing the outputs of calls in $R_B$
    - follow data flow forward from call outputs

- We do not consider control-flow dependencies
  - would lead to include too much code (taint explosion problem)
• The slice $\phi$ is not precise

• General purpose utility functions are frequently included (i.e: string processing)
  ▪ may be from statically linked libraries (i.e: libc)
  ▪ genotype model would match against any binary that links to the same library

• Backwards slicing goes too far back: initialization and even unpacking routines are often included
  ▪ genotype model would match against any malware packed with the same packer
Filtering Techniques

• Exclusive instructions:
  - set of instructions that manipulate tainted data every time they are executed
  - utility functions are likely to be also invoked on untainted data

• Discard whitelisted code:
  - whitelist obtained from other tasks or execution of the same sample, that do not perform B
  - could also use foreign whitelist
    - i.e: including common libraries and unpacking routines
• The slice $\phi$ is not complete

• Auxiliary instructions are not included
  ▪ loop and stack operations, pointer arithmetic, etc

• Add instructions that cannot be executed without executing at least one instruction in $\phi$

• Based on graph reachability analysis on the intra-procedural Control Flow Graph (CFG)
Finding Dormant Functionality

A. Dynamic Behavior Identification

B. Extracting Genotypes Models

C. Finding Dormant Functionality

$O'$  $B_1$  $B_2$  $B_3$  $B_n$

Malware sample  Behaviors

Genotype models

New malware samples
• Genotype is a set of instructions

• Use techniques by Kruegel et al. to efficiently match a binary against a set of genotype models

• Genotype model is its colored CFG
  ▪ nodes colored based on instruction classes

• 2 models match if they share at least one K-Node subgraph (K=10)

• We use Anubis as a generic unpacker
• Are the results accurate?
  ▪ when REANIMATOR detects a match, is there really the dormant behavior?
  ▪ how reliably does REANIMATOR detect dormant behavior in the face of recompilation or modification of the source code?

• Are the results insightful?
  ▪ does REANIMATOR reveal behavior we would not see in dynamic analysis?
Accuracy

- To test accuracy and robustness of our system we need a ground truth

- Dataset of 208 malware samples with source code
  - thanks to J. Oberheide and M. Bailey from U. Michigan

- Extract 6 genotype models from 1 sample

- Match against remaining 207 binaries
Accuracy

- Even with source, manually verifying code similarity is time-consuming

- Use a source code plagiarism detection tool
  - MOSS

- We feed MOSS the source code corresponding to each of the 6 behaviors
  - match it against the other 207 sources
  - MOSS returns a similarity score in percentage

- We expect REANIMATOR to match in cases where MOSS returns high similarity scores
MOSS Comparison

Percentage of matching code according to MOSS

Number of <behavior,bot> pairs

0-5  6-10  11-15  16-20  21-25  ...  76-80  81-85  86-90  91-95  96-100

MOSS
Reanimator
MOSS Comparison

Potential False Negatives

Potential False Positives

Number of <behavior,bot> pairs

Percentage of matching code according to MOSS
Accuracy Results

• We manually investigated the potential false positives and false negatives

• Low false negative rate (~1.5%)
  ▪ mostly small genotypes

• No false positives
  ▪ genotype model match always corresponds to presence of code implementing the behavior

• Also no false positives against dataset of ~2000 benign binaries
  ▪ binaries in system32 on a windows install
Robustness results when re-compiling same source

- Robust against different compilation options (<7% false negatives)
- Robust against different compiler versions
- Not robust against completely different compiler (>80% false negatives)
- Some robustness to malware metamorphism was demonstrated by Kruegel in a previous work
In-the-Wild Detection

- 10 genotype models extracted from 4 binaries
- 4 datasets
  - irc_bots: 10238 IRC bots
  - packed_bots: 4523 packed IRC bots
  - pushdo: 77 pushdo binaries (dropper, typically cutwail)
  - allaple: 64 allaple binaries (network worm)

- Reanimator reveals a lot of functionality not observed during dynamic analysis

_Warfare is the Way of deception. Therefore, if able, appear unable; if active, appear inactive; if near, appear far; if far, appear near._

Sun-tzu
In-the-Wild Detection

<table>
<thead>
<tr>
<th>Genotype</th>
<th>Phenotype</th>
<th>( \text{irc_bots} )</th>
<th></th>
<th>( \text{packed_bots} )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( B )</td>
<td>( S )</td>
<td>( D )</td>
<td>( B \cap S )</td>
</tr>
<tr>
<td>httpd</td>
<td>backdoor</td>
<td>2014</td>
<td>636</td>
<td>635</td>
<td>279</td>
</tr>
<tr>
<td>keylog</td>
<td>keylog</td>
<td>0</td>
<td>293</td>
<td>254</td>
<td>0</td>
</tr>
<tr>
<td>killproc</td>
<td>killproc</td>
<td>0</td>
<td>400</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>simplesam</td>
<td>spam</td>
<td>154</td>
<td>409</td>
<td>409</td>
<td>0</td>
</tr>
<tr>
<td>udpflood</td>
<td>packetflood</td>
<td>0</td>
<td>374</td>
<td>342</td>
<td>0</td>
</tr>
<tr>
<td>sniff</td>
<td>sniff</td>
<td>43</td>
<td>270</td>
<td>72</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Genotype</th>
<th>( \text{pushdo} )</th>
<th></th>
<th>( \text{allaple} )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( B )</td>
<td>( S )</td>
<td>( D )</td>
<td>( B \cap S )</td>
</tr>
<tr>
<td>drop</td>
<td>50</td>
<td>54</td>
<td>54</td>
<td>46</td>
</tr>
<tr>
<td>spam</td>
<td>1</td>
<td>43</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>scan</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rpcbind</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**B**: Behavior observed in dynamic analysis.
**S,D**: Functionality detected by Reanimator
• Tracking of malware evolution over time
• Let malware update and at each step:
  – Run malware in monitored environment to see behaviors
  – Identify the code changes responsible for malicious behavior changes
• Use the same techniques of REANIMATOR for identifying and labeling behaviors, and evolutions of binary code

Therefore, know the enemy's plans and calculate his strengths and weaknesses.
Provoke him, to know his patterns of movement.
Determine his position, to know the ground of death and of life.
Probe him, to know where he is strong and where he is weak.

Sun-tzu
Beagle: overview
### Beagle: some global results

<table>
<thead>
<tr>
<th>FAMILY_NAME</th>
<th>%TAGGED</th>
<th>%Labeled</th>
<th>%RATIO</th>
<th>%ADDED</th>
<th>%REMOVED</th>
<th>%SHARED</th>
<th>NEW</th>
<th>#LABELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>7.31 ± 1.70</td>
<td>6.68 ± 0.75</td>
<td>91.43</td>
<td>2.48 ± 2.96</td>
<td>2.83 ± 3.10</td>
<td>94.69 ± 3.75</td>
<td>176.2 ± 409.2</td>
<td>5</td>
</tr>
<tr>
<td>bot</td>
<td>32.36 ± 2.40</td>
<td>31.23 ± 2.95</td>
<td>96.50</td>
<td>10.59 ± 10.36</td>
<td>10.30 ± 10.42</td>
<td>79.11 ± 12.80</td>
<td>1361.4 ± 3937.2</td>
<td>11</td>
</tr>
<tr>
<td>gato</td>
<td>2.81 ± 1.22</td>
<td>1.15 ± 0.55</td>
<td>40.90</td>
<td>5.15 ± 5.14</td>
<td>5.57 ± 5.63</td>
<td>89.28 ± 7.48</td>
<td>2402.9 ± 7165.3</td>
<td>4</td>
</tr>
<tr>
<td>narue</td>
<td>15.90 ± 14.06</td>
<td>14.06 ± 13.40</td>
<td>88.42</td>
<td>12.08 ± 8.16</td>
<td>12.50 ± 9.32</td>
<td>75.41 ± 11.57</td>
<td>2500.1 ± 7747.2</td>
<td>12</td>
</tr>
<tr>
<td>genericDownloader</td>
<td>9.10 ± 1.93</td>
<td>8.58 ± 1.59</td>
<td>94.30</td>
<td>9.80 ± 9.85</td>
<td>9.58 ± 8.81</td>
<td>80.62 ± 12.48</td>
<td>3330.6 ± 7367.8</td>
<td>6</td>
</tr>
<tr>
<td>genericTrojan</td>
<td>22.94 ± 11.05</td>
<td>20.18 ± 10.69</td>
<td>87.97</td>
<td>16.66 ± 16.15</td>
<td>17.03 ± 15.15</td>
<td>66.31 ± 18.76</td>
<td>4974.1 ± 14339.6</td>
<td>11</td>
</tr>
<tr>
<td>ftor</td>
<td>12.66 ± 6.20</td>
<td>9.58 ± 4.70</td>
<td>75.70</td>
<td>6.47 ± 10.40</td>
<td>6.84 ± 9.96</td>
<td>86.69 ± 13.48</td>
<td>682.0 ± 1662.8</td>
<td>4</td>
</tr>
<tr>
<td>schos</td>
<td>24.20 ± 2.24</td>
<td>24.09 ± 2.26</td>
<td>99.53</td>
<td>5.18 ± 8.69</td>
<td>5.60 ± 10.10</td>
<td>89.23 ± 12.64</td>
<td>2145.3 ± 4065.3</td>
<td>12</td>
</tr>
<tr>
<td>mineGames</td>
<td>2.18 ± 0.30</td>
<td>1.96 ± 0.21</td>
<td>89.97</td>
<td>3.35 ± 3.12</td>
<td>3.37 ± 3.12</td>
<td>93.28 ± 5.44</td>
<td>420.0 ± 718.0</td>
<td>9</td>
</tr>
<tr>
<td>s</td>
<td>8.37 ± 2.59</td>
<td>6.15 ± 1.32</td>
<td>73.44</td>
<td>2.10 ± 2.24</td>
<td>3.59 ± 11.27</td>
<td>94.31 ± 11.28</td>
<td>1910.8 ± 6148.0</td>
<td>11</td>
</tr>
<tr>
<td>s</td>
<td>8.26 ± 1.56</td>
<td>6.44 ± 1.14</td>
<td>78.00</td>
<td>3.65 ± 3.07</td>
<td>5.25 ± 11.85</td>
<td>91.09 ± 12.41</td>
<td>4086.0 ± 11936.3</td>
<td>12</td>
</tr>
<tr>
<td>s</td>
<td>10.45 ± 2.67</td>
<td>7.91 ± 2.49</td>
<td>75.73</td>
<td>2.61 ± 2.20</td>
<td>4.51 ± 12.64</td>
<td>92.88 ± 12.47</td>
<td>2234.5 ± 7117.9</td>
<td>11</td>
</tr>
<tr>
<td>s</td>
<td>8.55 ± 2.15</td>
<td>6.53 ± 1.19</td>
<td>76.41</td>
<td>2.55 ± 2.51</td>
<td>3.93 ± 11.26</td>
<td>93.52 ± 11.35</td>
<td>2013.6 ± 6874.5</td>
<td>12</td>
</tr>
</tbody>
</table>
Beagle: breakdown of changes on behaviors

Gamarue family

Distribution of similarity

Bold line = median
Box = quantiles (0,25,75,100)
Circle = outlier
Beagle: some of the insights

• Changes and evolution
  – Some families are much more actively developed than others
  – Also we can pinpoint changes over individual behaviors, sometimes across the collection
  – In some cases, overall development appears constant/low, but we can disaggregate it to significant changes

• Effort
  – We have blocks in ASM, not LoC in source, but we can do some estimate
  – We estimate that avg added code in Zeus over each variation is 140–280 LoC, with peaks up to 9,000
  – Roughly holds for other families but we are less certain
  – Significant effort of development in malware
Automatically extracting behaviors

- Defining behaviors manually
  - Labor-intensive
  - Only a small subset of behaviors can be defined
  - Biased by previous experience of experts
- What if we could **extract** (interesting) behavior specifications in an **automatic** way from a large collection of **untagged** malware?

> Measurements are derived from ground, quantities are derived from measurement, calculations are derived from quantities, comparisons are derived from calculations, and victories are derived from comparisons

Sun-Tzu
APIs must be significant for one another so we can exploit Data Flow Analysis

```c
hout = CreateFile(_T("File.txt"), GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0); -> 0x<handler>

buf = VirtualAlloc(NULL, pageSize, flags, PAGE_READWRITE ); -> 0x<address>

ReadFile(hout, buf, 40, 0, NULL);

CloseHandle(hout)
```
Clustering step

Malware Traces → DataFlow with fingerprints → Cluster
Adding a semantic to the model

Stackoverflow

Posts related to a specific API

Based on semantic, for example we are interested in posts in which there are tags like "windows", "winapi"
We are not interested in some tags like "python", "php" etc.

We weight the importance of posts according to those wl/bl tags

WL / BL

Tags extraction

API

Tag: importance measure
InternetOpen (szAgent: atlsys13.exe: 1)
InternetOpenUrl,MapMemRegion,connect,recv,s
end
'Banload_09af6de40ab414f41ba48b447345e75d'

<table>
<thead>
<tr>
<th>Position</th>
<th>Tag (hint)</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>http</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>proxy</td>
<td>13.5</td>
</tr>
<tr>
<td>3</td>
<td>ftp</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>file</td>
<td>6.8</td>
</tr>
<tr>
<td>5</td>
<td>mfc</td>
<td>6.8</td>
</tr>
<tr>
<td>6</td>
<td>post</td>
<td>6.2</td>
</tr>
<tr>
<td>7</td>
<td>internet</td>
<td>5.6</td>
</tr>
<tr>
<td>8</td>
<td>upload</td>
<td>5.6</td>
</tr>
<tr>
<td>9</td>
<td>file-download</td>
<td>4.5</td>
</tr>
<tr>
<td>10</td>
<td>arrays</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>download</td>
<td>3.9</td>
</tr>
<tr>
<td>12</td>
<td>rich-internet-application</td>
<td>3.3</td>
</tr>
<tr>
<td>13</td>
<td>networking</td>
<td>3.3</td>
</tr>
<tr>
<td>14</td>
<td>httprequest</td>
<td>2.8</td>
</tr>
<tr>
<td>15</td>
<td>httpwebrequest</td>
<td>2.2</td>
</tr>
<tr>
<td>16</td>
<td>internet-explorer</td>
<td>1.7</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
Evaluation: family independency
43 found behaviors over 45
Plug-in for IDA (in the works)
Conclusions: how does this fit?

• We have combined static and dynamic analysis to:
  – Tag and analyze dormant code in malware collections
  – Track the evolution of malware over time
  – Automatically extract and tag behaviors

• Future directions in this area:
  – Automatic triage of new samples
  – Revolutionizing malware analysis (from analysis of samples to analysis of behaviors)

• In the bigger picture:
  – Characterize the adversary based on its artifacts
  – Extract actionable intelligence on relevant security phenomena based on available data
  – Provide the security community with tools and methodologies to improve data-driven targeting of the security efforts
Thanks!

Send me feedback!  
stefano.zanero@polimi.it  
or tweet: @raistolo

Most of the work presented was/is joint work with:

- UCSB: Christopher Kruegel, Martina Lindorfer
- Lastline: Paolo Milani Comparetti
- Northeastern University: Engin Kirda
- Politecnico di Milano: Mario Polino, Federico Maggi, Alessandro di Federico, Andrea Scorti
- TU Darmstadt: Guido Salvaneschi

Of course, errors and opinions are mine solely :-)

Stefano Zanero