Behavior-based Methods for Automated, Scalable Malware Analysis

Stefano Zanero, PhD
Assistant Professor, Politecnico di Milano
“He will win who knows when to fight and when not to fight... He will win who, prepared himself, waits to take the enemy unprepared. Hence the saying: If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.” [Sun-Tsu]
The analysis issue

Analysts are way too few, code is way too much

Need better ways to

- Automatically analyze/reverse engineer malware
- Automatically classify/cluster malware, e.g. in families

For both, we have two approaches with symmetric issues
<table>
<thead>
<tr>
<th>Static approaches</th>
<th>Dynamic approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Complete analysis</td>
<td>- “Dormant” code</td>
</tr>
<tr>
<td>- Difficult to extract semantics</td>
<td>+ Easy to see “behaviors”</td>
</tr>
<tr>
<td>- Obfuscation / packing</td>
<td>+ Malware unpacks itself</td>
</tr>
</tbody>
</table>
Turn weakness into strength, and strength into weakness, as Sun-Tzu would suggest: **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
Run malware in monitored environment and detect a malicious behavior \textit{(phenotype)}

Identify and model the code responsible for the malicious behavior \textit{(genotype model)}

Match genotype model against other unpacked binaries
A. Dynamic Behavior Identification

$O'$ → $B_1$ → $B_2$ → $B_3$ → ... → $B_n$

B. Extracting Genotypes Models

$\gamma_1 \leftarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_n$

C. Finding Dormant Functionality

New malware samples
Dynamic Behavior Identification

A. Dynamic Behavior Identification

$O' \rightarrow B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \cdots \rightarrow B_n$

Malware sample → Behaviors

B. Extracting Genotypes Models

Genotype models

C. Finding Dormant Functionality

New malware samples
Dynamic Behavior Identification

Run malware in instrumented sandbox
  ▪ Anubis (anubis.isclab.org)

Dynamically detect a behavior $B$ (*phenotype*)

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

$R_B$ is the 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

$O'$

Malware sample

Behaviors

Genotype models

New malware samples
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.
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 including 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
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

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

Malware sample \rightarrow Behaviors

\[ \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_n \]
Genotype models

New malware samples
Genotype is a set of instructions

Genotype model is its colored control flow graph (CFG)
- nodes colored based on instruction classes

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

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

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

Dataset of 208 malware samples with source code
  • thanks to Jon Oberheide and Michael Bailey from University of Michigan

Extract 6 genotype models from 1 sample

Match against remaining 207 binaries
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

The diagram shows the number of <behavior,bot> pairs for different percentage ranges of matching code according to MOSS. The x-axis represents the percentage ranges, and the y-axis represents the number of pairs. Two categories are shown: MOSS (blue) and Reanimator (red). The bar for MOSS is consistently higher than that for Reanimator in all the percentage ranges shown.
MOSS Comparison

Potential False Negatives

Potential False Positives
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 metamorphosis was demonstrated by Kruegel in a previous work
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 drops spam engine cutwail)
- allaple: 64 allaple binaries (network worm)

Reanimator reveals a lot of functionality not observed during dynamic analysis
# In-the-Wild Detection

**B**: Behavior observed in dynamic analysis.

**S, D**: Functionality detected by Reanimator

<table>
<thead>
<tr>
<th>Genotype</th>
<th>Phenotype</th>
<th>irc_bots</th>
<th></th>
<th>packed_bots</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>B ∩ 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>simplespam</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>pushdo</th>
<th></th>
<th>allaple</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>B ∩ 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
Beagle: overview
Beagle: how do we define a behavior

- We make use of an Anubis-like sandbox to automatically analyze system level activity.
- We extract automatically graphs of connected actions that we call (unlabeled) behaviors.
- We then label (some of) them manually, and can recognize with simple rules them across different samples.
- This is similar to the REANIMATOR behavior signatures.
- Opposed to REANIMATOR we tag code with behavior at a function-level granularity.
Table 1: Dataset. The labels in the first columns are based on Microsoft AV naming convention. The MD5 column is the number of distinct binaries encountered. Lifespan is the duration in days of the interval in which an MD5 was observed (min/max/mean/stddev).
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>Banload</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>Cycbot</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>Dupato</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>Gamarue</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>Grafftor</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>Kelihos</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>OnlineGames</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>Zeus</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>Zeus</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>Zeus</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>Zeus</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>
<tr>
<td>Zeus</td>
<td>8.82 ± 1.79</td>
<td>7.73 ± 1.36</td>
<td>87.65</td>
<td>3.12 ± 2.78</td>
<td>4.57 ± 11.33</td>
<td>92.32 ± 11.46</td>
<td>3245.9 ± 7456.3</td>
<td>12</td>
</tr>
<tr>
<td>Zeus</td>
<td>7.44 ± 1.31</td>
<td>6.41 ± 0.88</td>
<td>86.06</td>
<td>2.24 ± 2.51</td>
<td>4.53 ± 13.46</td>
<td>93.23 ± 13.46</td>
<td>2523.9 ± 6834.9</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 2: Overall tagged and labeled code (in each version), added, removed, shared code (between consecutive versions), and new code (with respect to all previous versions) for each family (mean ± variance, measured in basic blocks). #Labels is the number of distinct behavior labels detected throughout the versions.
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
Conclusions

- Structural analysis alone is too time and brain consuming
- Dynamic analysis alone has too many blind points
- We can combine both to obtain:
  - Dormant code analysis and tagging
  - Evolution tracking
  - Triage of new samples
- In the next slide pack I will present:
  - Automatic identification of behaviors
  - (hopefully) better means of classifying specimens in families
stefano.zanero@polimi.it
@raistolo

Most of the work presented was/is joint work with:
UCSB – Christopher Kruegel
Lastline – Paolo Milani Comparetti
Northeastern University – Engin Kirda
Technical University of Vienna – Martina Lindorfer
Politecnico di Milano - Federico Maggi, Alessandro di Federico,
Guido Salvaneschi, Mario Polino, Andrea Scorti

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

Research partially funded by the European Commission under FP7 project SysSec
Jackdaw
Automatic, unsupervised, scalable extraction and semantic tagging of (interesting) behaviors

Mario Polino, Andrea Scorti, Federico Maggi, Stefano Zanero

Politecnico di Milano
Dipartimento di Eletronica, Informazione e Bioingegneria
NECSTLab

Waseda university, Nov. 25th 2014
Pivot concept: behavior

behavior = sequence of actions = sequence of API calls (on Win binaries)

- download_execute
- recv
- WriteFile
- CreateProcess

Receives data from a network socket
Write a file
Create a new process
Defining Behaviors

Previous work: manual specification of behaviors

- Labor-intensive
- Only a small subset of behaviors can be defined manually
- Biased by previous experience of experts

Objective

Extract (interesting) behavior specifications in an automatic way from a large collection of (untagged) malware

Why?

Support the analyst by providing a list of important behaviors, with a rough explanation, to prioritize the analysis.
Our Approach: Jackdaw
System Architecture

Malware Instances

Malware Families

Step 1: Data Collection
Step 2: Clustering
Step 3: Behavior Extraction
Step 4: Semantic Tagging

Behaviors
First step: Data gathering

1. Dynamic Analysis: data flow analysis
   - API functions name
   - Parameters of API functions

2. Static Analysis: fingerprint of code associated to data flow
   - sub-graphs of the CFG
   - can be hashed and matched
   - reasonably resilient to polymorphism
Control Flow Graph Fingerprinting

**Static Analysis.**
Identify portions of CFG likely to come from the same source code.

Properties:
- Unique
- Robust to insertion / deletion
- Robust to modification

Both CFGs share a subgraph of given order
First Step: Data Gathering - Data cleaning

- Static data cleaning: remove the fingerprints of benign binaries (e.g., Windows libraries and exe)
- Dynamic data cleaning (Windows API name Normalization):
  
  **Prefixes**
  
<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSA</td>
<td>( \rightarrow ) socket</td>
</tr>
</tbody>
</table>

  **Suffixes**
  
<table>
<thead>
<tr>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateEventA, CreateEventW</td>
</tr>
</tbody>
</table>
Second Step: Clustering

Goal: Build clusters of similar data flows
Second step: Clustering

- Clustering of data flow in malware
- Feature: fingerprint
- simple one pass algorithm
- Threshold
- Similarity metrics
  
  \[
  J(A, B) = \frac{|A \cap B|}{|A \cup B|}
  \]
Third Step: Behavior Extraction

Goal: find API functions that represent each cluster (behavior model).
Third step: Behavior Extraction - MFR Heuristic

MFR Heuristic (Most Frequent Rule)

Model = API functions that appear often. How often? We set a threshold.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>API</th>
<th>NtClearEvent</th>
<th>CreateEvent</th>
<th>NtSetEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Flow 1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Data Flow 2</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Data Flow 3</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Data Flow 13</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>Data Flow 14</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Behavior Specification: \(NtClearEvent \land CreateEvent\)
Fourth Step: Semantic Tagger

Use **Crawler** to get knowledge and build significant **tag** for **behaviors**

For Each behavior:

- **Stackoverflow crawler**
  - 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

- **Posts related to a specific API**
- **Whitelist / BL**
- **Tags extraction**

API Tag : importance measure
Fourth step: Semantic Tagger

- We look for tags searching API function name, each element of powerset of API function in a model.
- Compute a score for each tag (based on post relevance and frequency of tags in post related to the search).
- Build a ranking of tags.
System Evaluation
The dataset:
- 1,272 samples from 17 malware families
Evaluation of behavior extraction: approach

- **Unsupervised learning** (no ground truth)
- We built a pseudo **ground truth**, asking experts to **manually** describe a **model**.
- We compare these **manually defined behavior models** with **behavior models automatically identified** by Jackdaw.
Evaluation of behavior extraction: results

Correctness:

Ground-truth Behavior:
firewall_settings

ShellExecute (advfirewall firewall add rule name: 1)

Automatic Behavior
ShellExecute (advfirewall firewall add rule name: 1)
RegOpenKey (hksui-1-5-21-842925246-1425521274-308236825-500\software\microsoft\internet explorer\main)
GetProcAddress

Completeness:
34 over 45 behavior models manually created by experts have been identified also by Jackdaw.
**Empirical** evaluation.

Example of behavior HTTP connection:

```plaintext
InternetOpen (szAgent: atlsys13.exe: 1)
InternetOpenUrl,MapMemRegion,connect,recv,send
(szUrl: http://robertokunihira.sites.uol.com.br/nordeste.jpg, ForeignPort:
['80']: 1, LocalAddress: ('tcp', public, ['1029']): 1, ForeignIP: public: 1)
'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>=</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>=</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>
Recognizing behaviors in unknown malware
Conclusions
Limitations and Future works

Limitations:
- needs buckets of variants of each malware family
- analyzed malware needs to be unpacked

Future Works:
- Introduce sequence/time concept in behavior models
- NLP to improve semantic tagging
Third step: Behavior Extraction - PLR Heuristic

PLR Heuristic (Propositional Logic Rule)
Let \( T \) be a set of elements; given a set of elements \( L \subseteq P(T) \), the solution is all sets \( Q \subseteq P(T) \) such that:

- \( \forall l \in L, \forall q_i, q_j \in Q \) with \( q_i \neq q_j \), if \( q_i \subset l \) then \( q_j \cap l = \emptyset \)
- \( \forall l \in L, \exists! q \in Q, q \subset l \)

Behavior Specification:

\[(\text{AtomAPI}_1 \land \text{AtomAPI}_2) \oplus (\text{RegCloseKey} \land \text{NtKeyAPI}_1 \land \text{NtKeyAPI}_2)\]
Jackdaw:

- **Automatically** extracts behavior models of widespread behaviors, exploiting both dynamic and static analysis.
- Assigns a set of **semantic tags** to each model to help analyst
- **Maps** behavior model on **binary code**, building a catalog of implementations of same behavior which can be used to attribute to family/group