Scalable, Behavior-Based Malware Clustering

Ulrich Bayer
Paolo Milani Comparetti
Clemens Hlauschek
Engin Kirda
Christopher Krügel

Secure Systems Lab/TU Vienna
Eurecom
University of California, Santa Barbara

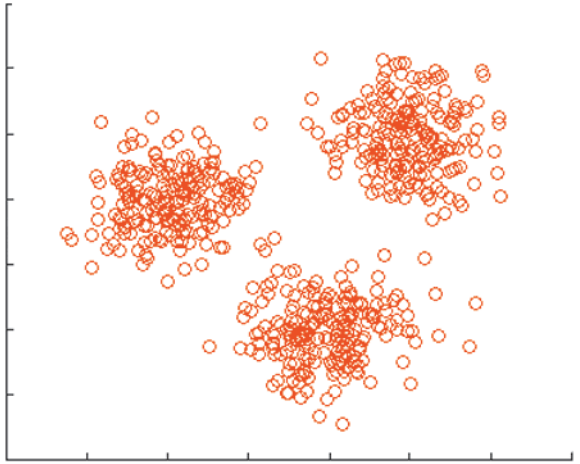
Motivation

- Thousands of new malware samples appear each day
- Automatic analysis systems allow us to create thousands of analysis reports
- Now a way to group the reports is needed. We would like to cluster them into sets of malware reports that exhibit similar behavior.
  - we require automated clustering techniques
- Clustering allows us to:
  - discard reports of samples that have been seen before
  - guide an analyst in the selection of those samples that require most attention
  - derive generalized signatures, implement removal procedures that work for a whole class of samples

Scalable, Behavior-Based Malware Clustering

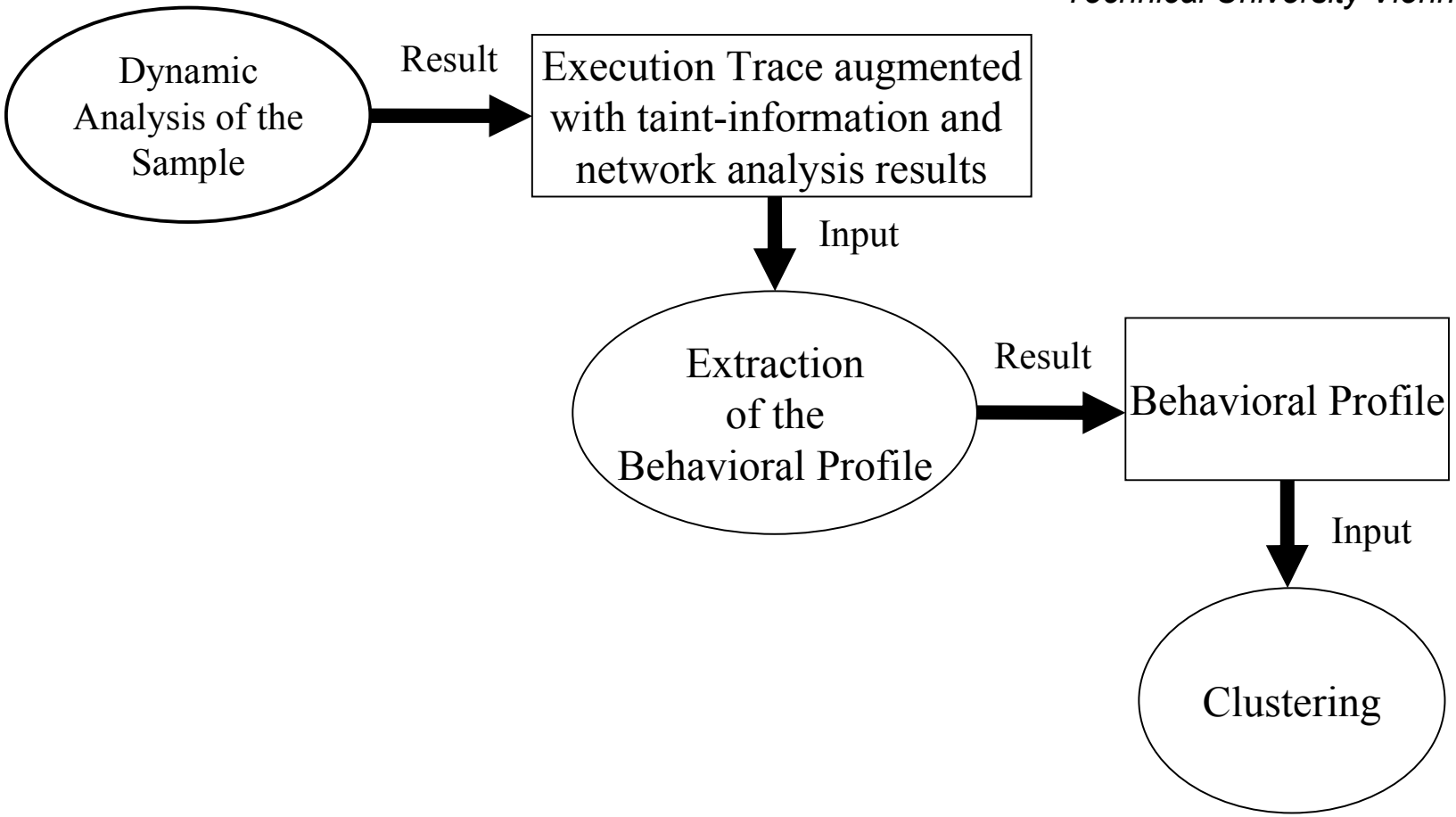
Secure Systems Lab
Technical University Vienna

- **Malware Clustering:** Find a partitioning of a given set of malware samples into subsets so that subsets share some common traits (i.e., find "virus families")



- **Behavior-Based:** A malware sample is represented by its actions performed at run-time
- **Scalable:** It has to work for large sets of malware samples

System Overview



Dynamic Analysis

- Based on our existing automatic, dynamic analysis system called Anubis
  - Anubis is a full-system emulator
  - Anubis generates an execution trace listing all invoked system calls
- In this work, we extended Anubis with:
  - system call dependencies (Tainting)
  - control flow dependencies
  - network analysis (for accurately describing a sample's network behavior)
- Output of this step: Execution trace augmented with taint information and network analysis results

Extraction Of The Behavioral Profile

- In this step, we process the execution trace provided by the ‘dynamic analysis’ step

- Goal: abstract from the system call trace
  - system calls can vary significantly, even between programs that exhibit the same behavior
  - remove execution-specific artifacts from the trace

- A behavioral profile is an abstraction of the program's execution trace that accurately captures the behavior of the binary

Reasons For An Abstract Behavioral Description

- Different ways to read from a file:

A:
```
f = fopen("C:\\test");
read(f, 1);
read(f, 1);
read(f, 1);
```

B:
```
f = fopen("C:\\test");
read(f, 3);
```

- Different system calls with similar semantics
  - e.g., NtCreateProcess, NtCreateProcessEx
- You can easily interleave the trace with unrelated calls:

C:
```
f = fopen("C:\\test");
read(f, 1);
readRegValue(..);
read(f, 1);
```

Reasons For An Abstract Behavioral Description

- Different ways to read from a file:

A:
```
f = fopen("C:\\test");
read(f, 1);
read(f, 1);
read(f, 1);
```

B:
```
f = fopen("C:\\test");
read(f, 3);
```

- Different system calls with similar semantics
  - e.g., NtCreateProcess, NtCreateProcessEx
- You can easily interleave the trace with unrelated calls:

C:
```
f = fopen("C:\\test");
read(f, 1);
readRegValue(..);
read(f, 1);
```

Reasons For An Abstract Behavioral Description

- Different ways to read from a file:

A:
```
f = fopen("C:\\test");
read(f, 1);
read(f, 1);
read(f, 1);
```

B:
```
f = fopen("C:\\test");
read(f, 3);
```

- Different system calls with similar semantics
  - e.g., NtCreateProcess, NtCreateProcessEx
- You can easily interleave the trace with unrelated calls:

C:
```
f = fopen("C:\\test");
read(f, 1);
readRegValue(..);
read(f, 1);
```

Reasons For An Abstract Behavioral Description

- Different ways to read from a file:

A:
```
f = fopen("C:\\test");
read(f, 1);
read(f, 1);
read(f, 1);
```

B:
```
f = fopen("C:\\test");
read(f, 3);
```

- Different system calls with similar semantics
  - e.g., NtCreateProcess, NtCreateProcessEx
- You can easily interleave the trace with unrelated calls:

C:
```
f = fopen("C:\\test");
read(f, 1);
readRegValue(..);
read(f, 1);
```

Elements Of A Behavioral Profile

- OS Objects: represent a resource such as a file that can be manipulated via system calls
  - has a name and a type
- OS Operations: generalization of a system call
  - carried out on an OS object
  - the order of operations is irrelevant
  - the number of operations on a certain resource does not matter
- Object Dependencies: model dependencies between OS objects (e.g., a copy operation from a source file to a target file)
  - also reflect the true order of operations
- Control Flow Dependencies: reflect how tainted data is used by the program (comparisons with tainted data)

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read – mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read – mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read – mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read - mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read – mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read - mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read - mem_write
```

Example: Behavioral Profile


```
src = NtOpenFile("C:\\sample.exe");
// memory map the target file
dst = NtCreateFile("C:\\Windows\\" + GetTempFilename());
dst_section = NtCreateSection(dst);
char *base = NtMapViewOfSection(dst_section);
while(len < length(src)) {
  *(base+len)=NtReadFile(src, 1); len++; }
```

```
Op|File|C:\sample.exe
  open:1, read:1
Op|File|RANDOM_1
  create:1
Op|Section|RANDOM_1
  open:1, map:1, mem_write: 1
Dep|File|C:\sample.exe -> Section|RANDOM_1
  read - mem_write
```

Scalable Clustering

- Most clustering algorithms require to compute the distances between all pairs of points => $O(n^2)$
- We use LSH (locality sensitive hashing), a technique introduced by Indyk and Motwani, to compute an approximate clustering that requires less than n^2 distance computations
- Our clustering algorithm takes as input a set of malware samples where each malware sample is represented as a set of features

  $\Rightarrow$ we have to transform each behavioral profile into a feature set first
- Our similarity measure: Jaccard Index defined as

$$J(a,b) = |a \cap b| / |a \cup b|$$

LSH Clustering

- We are performing an approximate, single-linkage hierarchical clustering:
- Step 1: Locality Sensitive Hashing
  - to cluster a set of samples we have to choose a similarity threshold t
  - the result is an approximation of the true set of all near (as defined by the parameter t) pairs
- Step 2: Single-Linkage hierarchical clustering

Evaluating Clustering Quality

- For assessing the quality of the clustering algorithm, we compare our clustering results with a reference clustering of the same sample set
  - since no reference clustering for malware exists, we had to create it first
- Reference Clustering:
  1. we obtained a random sampling of 14,212 malware samples that were submitted to Anubis from Oct. 27th 2007 to Jan. 31st 2008
  2. we scanned each sample with 6 different virus scanners
  3. we selected only those samples for which the majority of the anti-virus programs reported the same malware family. This resulted in a total of 2,658 samples.
  4. we manually corrected classification problems

Quantitative Evaluation

- We ran our clustering algorithm with a similarity threshold t = 0.7 on the reference set of 2,658 samples.
- Our system produced 87 clusters while the reference clustering consists of 84 clusters.

- Precision: 0.984
  - precision measures how well a clustering algorithm distinguishes between samples that are different
- Recall: 0.930
  - recall measures how well a clustering algorithm recognizes similar samples

Comparative Evaluation

Behavioral Description	Similarity Measure	Clustering	Optimal Threshold	Quality
Bailey-profile	NCD	Exact	0.75	**0.916**
Bailey-Profile	Jaccard Index	Exact	0.63	**0.801**
Syscalls	Jaccard Index	Exact	0.19	**0.656**
Our Profile	Jaccard Index	Exact	0.61	**0.959**
Our Profile	Jaccard Index	LSH	0.60	**0.959**

Performance Evaluation

- Input: 75,692 malware samples

- Previous work by Bailey et al (extrapolated from their results of 500 samples):
  Number of distance calculations: 2,864,639,432
  Time for a single distance calculation: 1.25 ms
  Runtime: 995 hours (~ 6 weeks)
- Our results:
  Number of distance calculations: 66,528,049
  Runtime: 2h 18min

Conclusions

- Novel approach for clustering large collections of malware samples
  - dynamic analysis
  - extraction of behavioral profiles
  - clustering algorithm that requires less than a quadratic amount of distance calculations
- Experiments on real-world datasets that demonstrate that our techniques can accurately recognize malicious code that behaves in a similar fashion

- Available online: http://anubis.iseclab.org