Utilizing Cyber Armsraces for the Good Guys

Nur Zincir-Heywood
zincir@cs.dal.ca
http://www.cs.dal.ca/~zincir/
Dal NIMS Lab: Cybersecurity and ML

Cyber
Networks /Applications /Services
Systems that can Adapt
Identify Different Behaviours
Cyber Security: Bad vs Good

An Artificial Arms Race: Could it Improve Mobile Malware Detectors?
The State of Malware

- Malware can be easily modified
- A malware detector may see the problem like this

Modified malware type A to look different
The State of Malware

• Detectors must adapt but also be proactive as the “wild” changes

Modified malware type A to look different

Known malware type A

Known malware type B
Arms Race

• This is a competition between attackers (malware) and defenders (malware detectors)

© Warner Bros, 1948
ArmsRace

• We can create the modified malware to be ready
Malware on Android

• Android malware
  • Many categories with different examples

• Format
  • Modified versions of non malware apps
  • Similar Android permission request combinations
Malware detection on Android

• Identifiable by
  • Permissions Requested
  • Code Features

• Machine Learning
  • Could be a good match
  • 15 to 20 features were effective in past research
Malware Detector Implementation

15 permission features known (and tested) to be related to malware behaviours

- Internet
- Read SMS
- Write SMS
- Read contacts
- Read external storage
- Write external storage
- Install Packages
- Admin
- Accessibility services
- On Boot
- Phone information
- Camera
- Microphone
- Calendar
- GPS
Malware Detector Implementation

8 code features counting known (and tested) to be related to malware behaviours

- Classes
- Classes using interfaces
- Classes containing annotations
- Direct methods
- Virtual methods
- Abstract methods
- Class level Static variables
- Class level Instanced variables
Training Workshop for Educators and Network Engineers on High Speed Network Protocols and Security 2020

ARMSRACE
Malware Sources

• We tested malware from
  • Drebin and Genome
    • datasets of malware samples
    • 700 apps used for static training set
    • 300 apps used for validation
  • Co-evolved
    • For testing a subsample of 10000 generated apps were collected
• GetJar
  • An app store where malware was found
Benign Sources

• We use 2 sources:
  • Fdroid and G-Play
    • widely used open source app stores
• 700 apps of each used for training
• 300 apps of each used for testing
### 100 generation F-Droid & Google Play

<table>
<thead>
<tr>
<th>Type</th>
<th>GA</th>
<th>Co-evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>41</td>
<td>30</td>
</tr>
<tr>
<td>Features used</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Precision F-Droid</td>
<td>97.0%</td>
<td>97.5%</td>
</tr>
<tr>
<td>Precision Google Play</td>
<td>97.3%</td>
<td>97.8%</td>
</tr>
<tr>
<td>Recall on Generated Malware</td>
<td>54.5%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Out in the wild

• GetJar
  • Open Appstore
  • 3 million downloads a day
• Many apps look sketchy
  • All apps marked as malware were hard to detect malware
    • Using Virus total, collection of public malware detectors
    • Confirmed by
      • All apps
  • 9 of 11 correctly marked
• GetJar’s app was marked
  • Virus total considers this to be safe.
## Detection Rate on GetJar Apps

<table>
<thead>
<tr>
<th>Application</th>
<th>Virus Total</th>
<th>C5.0</th>
<th>GP</th>
<th>Arms race GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MicrowaveRecipes</td>
<td>31%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>God of war Wall paper</td>
<td>36%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Facebook Password Hacker</td>
<td>22%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Footcare salon</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Application</td>
<td>46%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Saavn_getjar</td>
<td>5%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>PS4 emulator</td>
<td>11%</td>
<td>100%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Subway Servers Hack and cheat</td>
<td>9%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Miss You - Whatsapp</td>
<td>24%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Cam Scanner License</td>
<td>27%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Most common features

<table>
<thead>
<tr>
<th>GP</th>
<th>Arms race GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_PHONE_STATE</td>
<td>READ_PHONE_STATE</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>Count of classes</td>
</tr>
<tr>
<td>INTERNET</td>
<td>Count of static variables</td>
</tr>
<tr>
<td>Count of abstract classes</td>
<td>READ_CALENDER</td>
</tr>
<tr>
<td>Count of static variables</td>
<td>SEND_SMS</td>
</tr>
<tr>
<td>Count of virtual methods</td>
<td>INTERNET</td>
</tr>
<tr>
<td>Count of instanced variables</td>
<td>Count of direct method calls</td>
</tr>
<tr>
<td>Count of classes</td>
<td>BIND_ACCESSIBILITY_SERVICE</td>
</tr>
<tr>
<td>RECEIVE_BOOT_COMPLETED</td>
<td>RECORD_AUDIO</td>
</tr>
<tr>
<td>Count of direct method calls</td>
<td>Count of Classes with interfaces</td>
</tr>
</tbody>
</table>
Darwinian Malware Detectors: Evolutionary Solutions to Android Malware

Zachary Wilkins (zachary.wilkins@dal.ca)
Nur Zincir-Heywood (zincir@cs.dal.ca)

Dalhousie University
Halifax, NS, Canada

ACM SecDef 2019
Principles

- Simulate a competition between malware and detectors
- Evolved malware are a prediction of future adversaries
- Focus on privacy leakage malware
- Detector Generation
  - Linear genetic programming
  - Sequence of instructions for virtual machine
  - Read / write memory, read input, math ops
  - Many individuals \(\rightarrow\) Gradient of feedback
Assemblyline

Principles

- Services tailored to certain file types
- Rank file from -1000 to 1000, benign to malware
  - Raise alert above 500
- Android service: APKaye
  - Disassemble APK and extract features
  - Check features against rule-base

- Services tailored to certain file types
- Rank file from -1000 to 1000, benign to malware
  - Raise alert above 500
- Android service: APKaye
  - Disassemble APK and extract features
  - Check features against rule-base
**VirusTotal**

**Principles**
- One system is good, 70+ systems is better!
- AVG, McAfee, Kaspersky, Symantec, TrendMicro...
- Aggregate results from detectors: more information
- Dynamic analysis also performed (if possible)

**Principles**
- Submit file or search hash, IP, URL...
- Primary entrypoints: web or API
- Basic API is free but limited
- No need for an account!
Android Malware Dataset (AMD)

- Academic dataset from University of South Florida
- 24,553 malware samples from 2010 to 2016
- 135 varieties from 71 families
CICAndMal2017 (UNB)

- Academic dataset from University of New Brunswick
- 426 malware and 1,700 benign from 2015 to 2017
- Four categories and 42 families
VirusShare

- Community dataset from anonymous donors
- 35,397 malware samples dated 2013 and 2014
- Over 10,000 corrupted files (impossible to decompile)
Experiment Setup

Overview of training / testing

[Diagram showing the process of training and testing with nodes labeled Train, Test, AR, MD, AL, VT, Predictions, Detection Rate, and False Positives. The diagram illustrates the flow of data from malicious and benign APKs to training and testing stages, leading to predictions and evaluation metrics.]
### VirusTotal Unknown Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Seen</th>
<th>Unseen</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>24553</td>
<td>0</td>
<td>49.50</td>
<td>19.30</td>
<td>81.97</td>
</tr>
<tr>
<td>UNB Ben</td>
<td>1700</td>
<td>0</td>
<td>99.99</td>
<td>99.84</td>
<td>100.00</td>
</tr>
<tr>
<td>UNB Mal</td>
<td>426</td>
<td>0</td>
<td>47.95</td>
<td>0.00</td>
<td>80.00</td>
</tr>
<tr>
<td>VirusShare</td>
<td>35397</td>
<td>0</td>
<td>53.34</td>
<td>0.00</td>
<td>82.09</td>
</tr>
</tbody>
</table>

- Malware avg. is consistent at ~ 50%.
- Benignware is nearly perfect
- Similar ordering in difficulty to other detectors
- UNB Malware and VirusShare have totally undetectable samples!
Assemblyline Unknown Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Is Malicious</th>
<th>Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>T</td>
<td>16850/23618 (71.34)</td>
</tr>
<tr>
<td>UNB Ben</td>
<td>F</td>
<td>1154/1688 (68.36)</td>
</tr>
<tr>
<td>UNB Mal</td>
<td>T</td>
<td>315/424 (74.29)</td>
</tr>
<tr>
<td>VirusShare</td>
<td>T</td>
<td>16095/20884 (77.07)</td>
</tr>
</tbody>
</table>

- Detection rates are ~15% lower than MOCDroid
- Detection rates are ~23% lower than ArmsRace!
- Poor benignware detection is a factor
  - 9% and 22% lower when UNB Benign removed
MOCDroid Unknown Results

<table>
<thead>
<tr>
<th>Unknown</th>
<th>Malware</th>
<th>Benign</th>
<th>Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>Drebin</td>
<td>F-Droid</td>
<td>21518/24553 (87.64)</td>
</tr>
<tr>
<td>AMD</td>
<td>Genome</td>
<td>F-Droid</td>
<td>15104/24553 (61.52)</td>
</tr>
<tr>
<td>UNB Ben</td>
<td>Genome</td>
<td>F-Droid</td>
<td>1675/1700 (98.53)</td>
</tr>
<tr>
<td>UNB Ben</td>
<td>Drebin</td>
<td>Google Play</td>
<td>1584/1700 (93.18)</td>
</tr>
<tr>
<td>UNB Mal</td>
<td>Drebin</td>
<td>Google Play</td>
<td>294/426 (69.01)</td>
</tr>
<tr>
<td>UNB Mal</td>
<td>Genome</td>
<td>F-Droid</td>
<td>153/426 (35.92)</td>
</tr>
<tr>
<td>VirusShare</td>
<td>Drebin</td>
<td>F-Droid</td>
<td>19775/20984 (94.24)</td>
</tr>
<tr>
<td>VirusShare</td>
<td>Genome</td>
<td>Google Play</td>
<td>15147/20984 (72.18)</td>
</tr>
</tbody>
</table>

- AMD, VirusShare do very well
- UNB Benign: excellent for every model
- No model is very good at UNB Malware
- Train on Drebin: 22% avg. increase
### ArmsRace Unknown Results

<table>
<thead>
<tr>
<th>Unknown</th>
<th>Malware</th>
<th>Benign</th>
<th>Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>Drebin</td>
<td>Google Play</td>
<td>24254/24449 (99.20)</td>
</tr>
<tr>
<td>AMD</td>
<td>Genome</td>
<td>Google Play</td>
<td>19020/24449 (77.79)</td>
</tr>
<tr>
<td>UNB Ben</td>
<td>Drebin</td>
<td>F-Droid</td>
<td>1642/1700 (96.59)</td>
</tr>
<tr>
<td>UNB Ben</td>
<td>Drebin</td>
<td>Google Play</td>
<td>1249/1700 (73.47)</td>
</tr>
<tr>
<td>UNB Mal</td>
<td>Drebin</td>
<td>Google Play</td>
<td>381/425 (89.65)</td>
</tr>
<tr>
<td>UNB Mal</td>
<td>Genome</td>
<td>F-Droid</td>
<td>269/425 (63.29)</td>
</tr>
<tr>
<td>VirusShare</td>
<td>Drebin</td>
<td>Google Play</td>
<td>20757/20972 (98.97)</td>
</tr>
<tr>
<td>VirusShare</td>
<td>Genome</td>
<td>Google Play</td>
<td>17680/20972 (84.30)</td>
</tr>
</tbody>
</table>

- **AMD, VirusShare** near perfect with Drebin / Google Play
- **UNB Benign:** still pretty good
- **Train on Drebin:** 12% avg. increase
- **UNB Malware with Drebin / Google Play** → Almost 90%!
RETURN
ORIENTED
PROGRAMME
EVOLUTION with
ROPER

Olivia Lucca Fraser ofraser@dal.ca
Nur Zincir-Heywood zincir@cs.dal.ca
Malcolm Heywood mheywood@cs.dal.ca
John T. Jacobs John_T_Jacobs@raytheon.com

NIMS Laboratory @ Dalhousie University
Raytheon Space & Airborne Systems
https://github.com/oblivia-simplex

ACM SecDef 2017
Questions:
- What is return oriented programming?
- How might evolutionary methods be applied to ROP?
- How do we best cultivate the evolution of ROP payloads?
- What sort of things are they capable of?
3. The Basic Idea

ROPER is a system for evolving populations of ROP-chains for a target executable.
A Quick Introduction to Return Oriented Programming

- SITUATION: You have found an exploitable vulnerability in a target process, and are able to corrupt the instruction pointer.
- PROBLEM: You can’t write to executable memory, and you can’t execute writeable memory. Old-school shellcode attacks won’t work.
- SOLUTION: You can’t introduce any code of your own, but you can reuse pieces of memory that are already executable. The trick is rearranging them into something useful.
What is a ROP gadget?

- A ‘gadget’ is any chunk of machine code that
  1. is already mapped to executable memory
  2. allows us to regain control of the instruction pointer after it executes

- since all we control is the data being read by the process, the only ‘gadgets’ useful to us are those that
  1. perform some helpful operation, and then
  2. alter the instruction pointer according to data we control

- ideally, each gadget will perform its operation, and then finish by sending the instruction pointer to the next gadget we want to make use of
Generalization of the Gadget Concept

- the precise meaning of a ‘return’ instruction is architecture-dependent; not all architectures implement return as a pop into PC (MIPS, e.g.)
- the essential idea we’re after is stack-controlled jumps
- this means we don’t need to limit our search to ‘return’s
- we can broaden it to include any sequence of instructions that culminates in a jump to a location that’s determined by the data on the stack
- this gives us what’s commonly called ‘JOP’, or jump-oriented programming
Uneven Raw Materials

Register usage in tomato-RT-N18U-httpd, an ARM router HTTP daemon

![Bar chart showing register usage in an ARM router HTTP daemon.](image)
4. Bird's-Eye View of ROPER

- Data or Pattern Script (3)
- Genetic Process (4)
- Virtual Machine (5)
- Gadget Extraction (2)
- Target Binary (1)

Arrows indicate the flow of information:
- From Data or Pattern Script to Genetic Process via fitness criteria.
- From Genetic Process to Virtual Machine via genotype.
- From Virtual Machine to Target Binary via phenotype.
- From Target Binary to Gadget Extraction via genetic material.

Note: The diagram illustrates the lifecycle of ROPER, highlighting the interconnections between its components.
17. Pattern matching

The most basic type of problem that ROPER can breed a population of chains to solve is that achieving a determinate register state in the CPU, specified by a simple pattern consisting of integers and wildcards.

This isn’t the most intriguing thing that ROPER can do, but it is fairly useful, automating the ordinary, human task of assembling a ROP chain that prepares the CPU for a system call - to spawn a process, write to a file, open a socket, etc.

For example, suppose we wanted to prime the CPU for the call

`execv("/bin/sh", ["/bin/sh"], 0);`

We’d need a ROP chain that sets r0 and r1 to point to some memory location that contains "/bin/sh", sets r2 to 0, and r7 to 11. Once that’s in place spawning a shell is as simple as jumping to any given address that contains an svc instruction.

One of ROPER’s more peculiar solutions to this problem - using gadgets from a Tomato router’s HTTP daemon - is on the next slide...
18. Example of a Compiled Shell-Popping ROP-chain (by ROPeMe, not ROPER)

Payload:
00002d38  deadbeef
0000bb3d  00000000  4b4e554b
000256f9  00000000  4b4e554b  4b4e554c
0000bb3d  00000000  00000000
00001804  4b4e554a  00000000

Runtime:
00002d38  pop {r0, pc}
0000bb3d  pop {r1, r7, pc}
000256f9  pop {r2, r3, r6, pc}
0000bb3d  pop {r1, r7, pc}
0001804  svc 0x0
0001808  pop {r4, r8}
000180c  bx lr

Source: Long Le, ARM Exploitation ROPMAP, Blackhat 2011
Table 5.2: Execution trace of a chain that generates the register pattern required for a call to `execv("/bin/sh", ["/bin/sh"], NULL)` in `tomato-RT-N18U-httpd`, by modifying its own call stack and executing numerous "stray" or "extended" gadgets, in the Poclure population. Modifications to the gadget stack are in red, jumps are in violet, and completion of target CPU pattern is in blue. Free branches are separated by blank lines. The final instruction jumps to the designated stop address, \texttt{0x000000000f}.
Classifying flowers using HTTP daemon ROP Chains – Detection Rate 96.6%

Figure 5.18: Map of the Iris dataset. Triangle points represent petal measurements, and square points represent sepal measurements, with length on the X-axis and width on the Y-axis. Colour maps to species: green for setosa, maroon for versicolor, and pink for virginica.

Figure 5.19: A very good run on the Iris classification task, employing the fitness sharing algorithm documented in §4.4.4 (Raspop population). The filled curves surrounding each mean difficulty line again represent the standard deviation of difficulty for that exemplar class.
What did we learn?

• Data driven
  • New insight and knowledge
• Input – representation
  • Traffic / Text / Usage
• Generalization
  • Time & Location & Evasion
• Output – objectives
  • Known behavior
  • Behaviour changed
  • Unknown / new behaviour
  • Value of certainty
How much prior knowledge?

Data and Objectives
More prior info → Constraints search space
More prior info → Creates Blind side

How much ground truth?

What is the cost of providing labels?
What is the deployment environment?
Location, Time, Evasion
What is next?

Ever changing cycle

“Always” Learning to model the “change”
Thank You 😊
Questions?

Web: https://web.cs.dal.ca/~zincir

Dal NIMS Lab: https://www.youtube.com/watch?v=dYWzpW1bqo