
MACHINE LEARNING & SECURITY RESEARCH

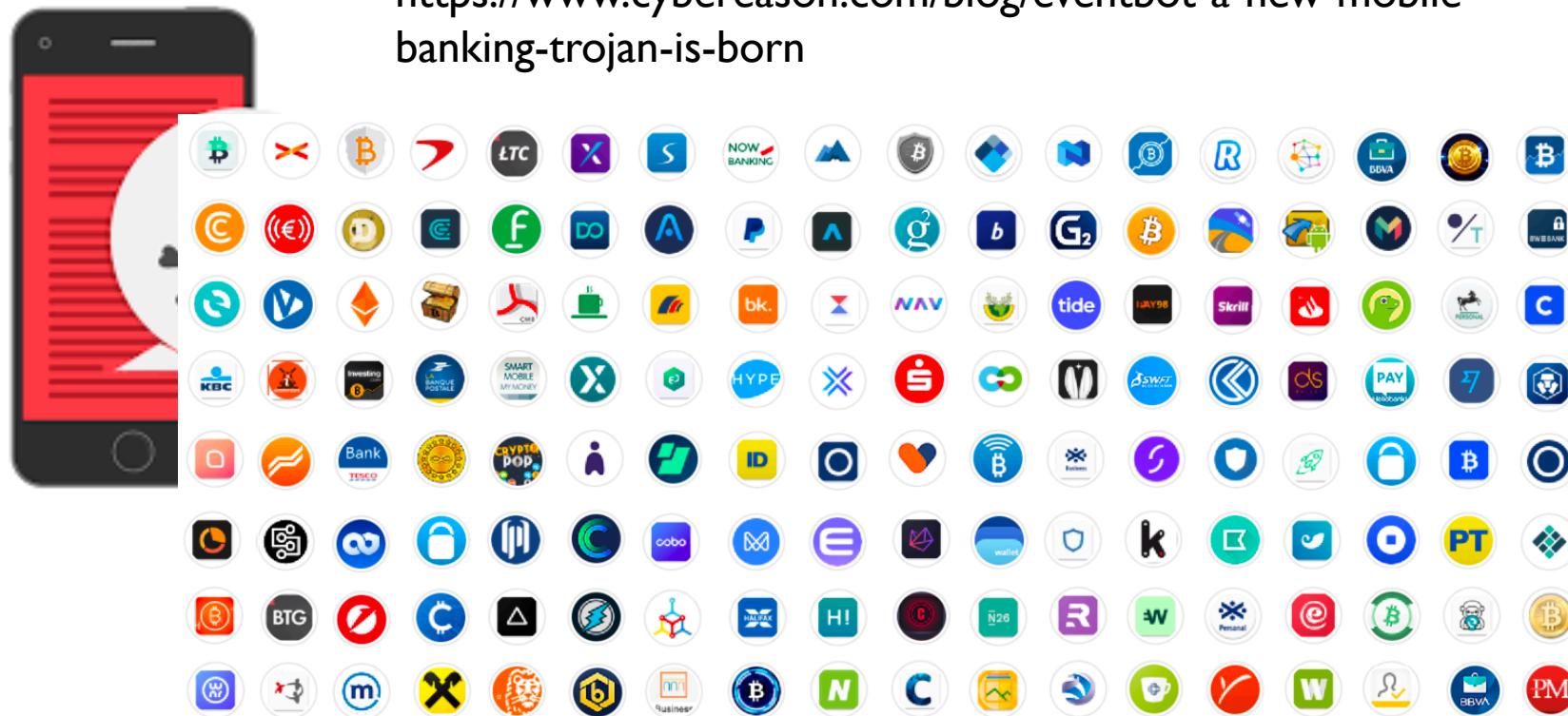
Giuseppe Antonio Di Luna

A DAY IN OUR LIFE



A PANOPLY OF THREATS

<https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born>

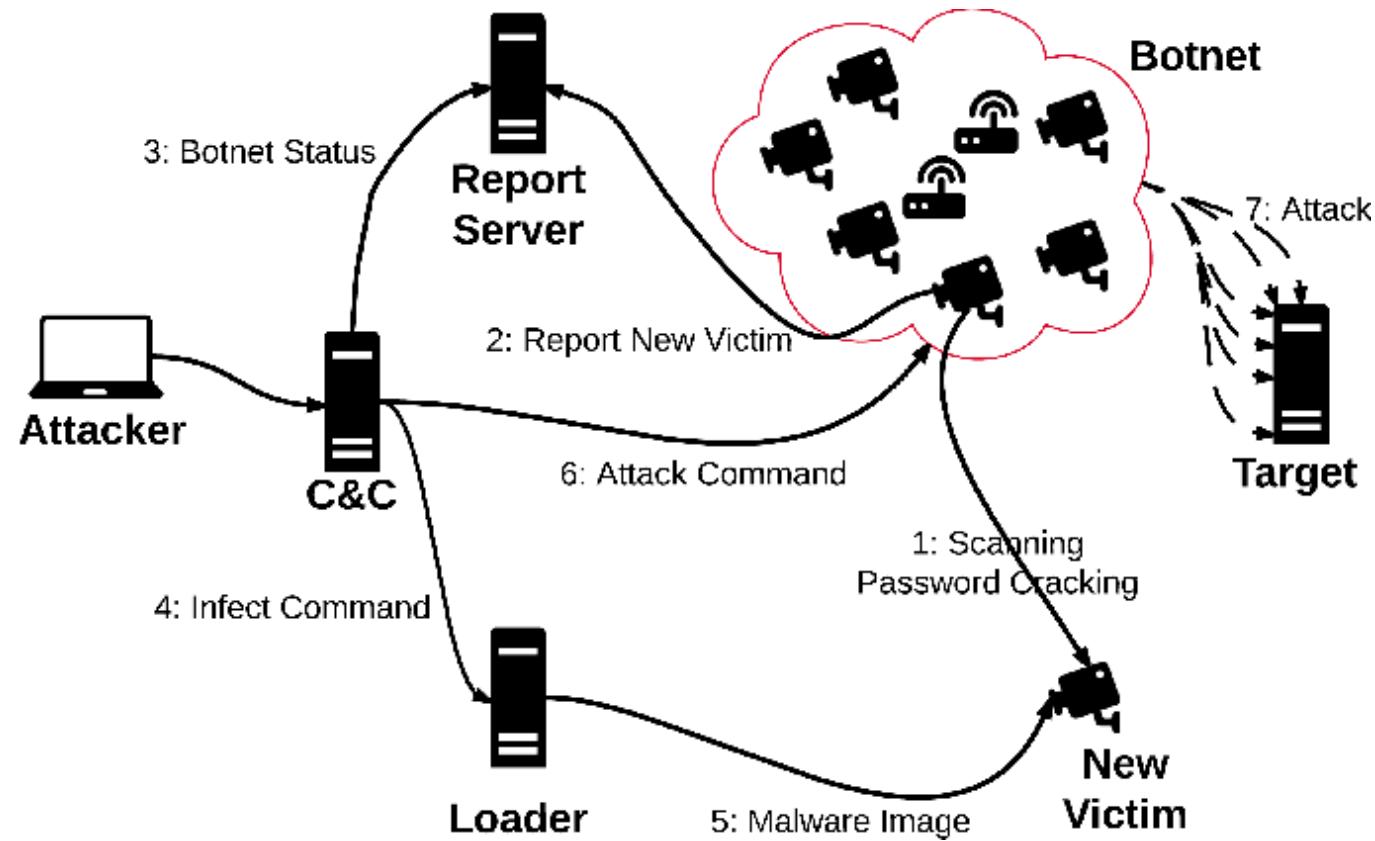


A PANOPLY OF THREATS



- Banking data
- Identity theft
- Surveillance

A PANOPLY OF THREATS



Mirai Botnet:
DDoS attack on 20 September 2016 on the [Krebs on Security](#) site which reached 620 Gbit/s. [Ars Technica](#) also reported a 1 Tbit/s attack on French web host [OVH](#).

A PANOPLY OF THREATS

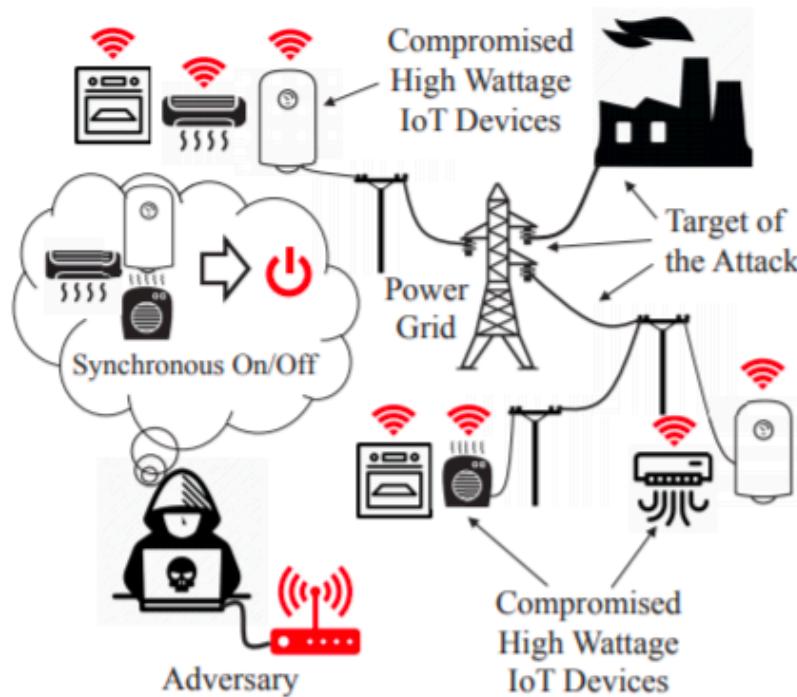
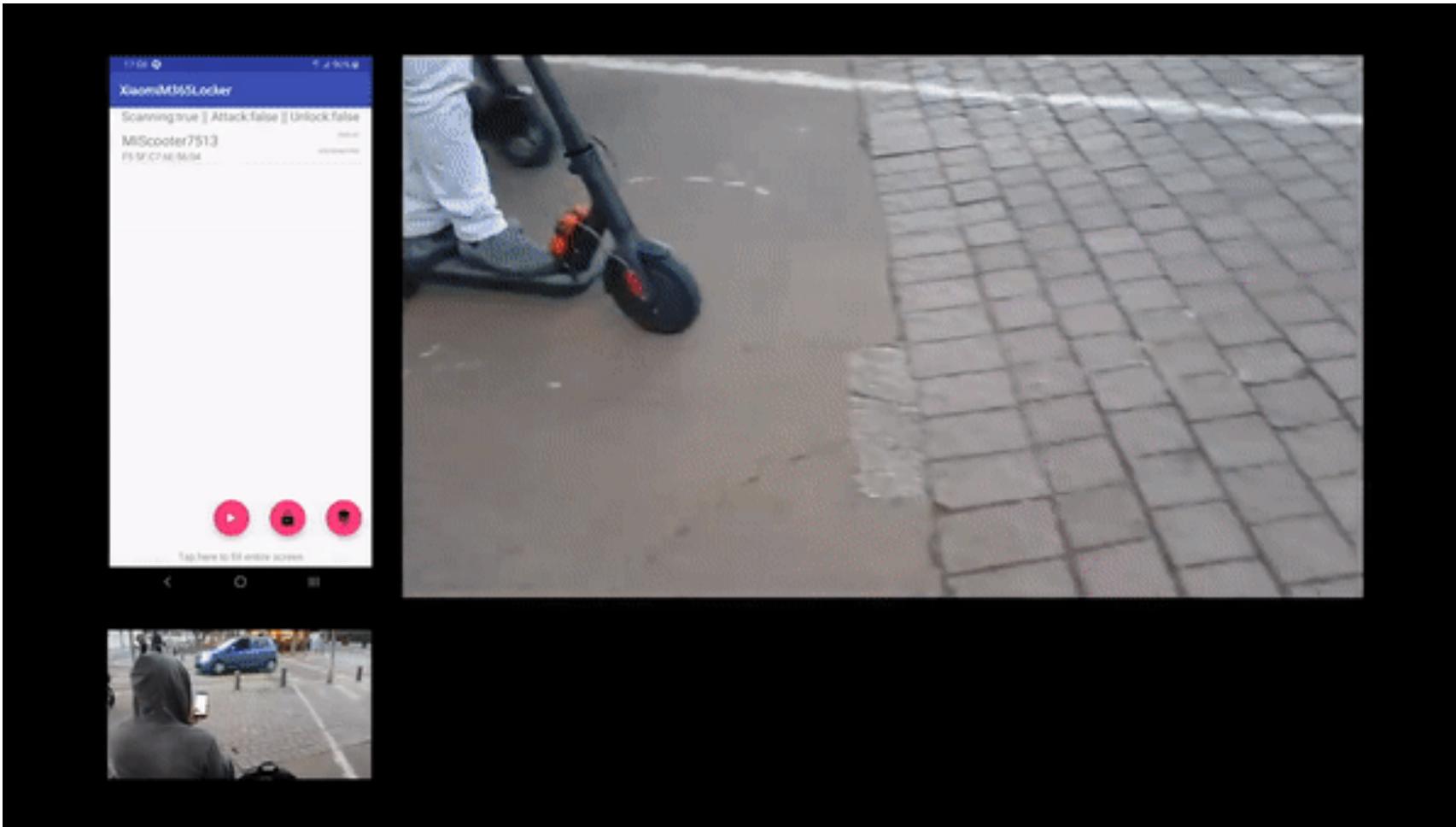


Figure 1: The MadIoT attack. An adversary can disrupt the power grid's normal operation by synchronously switching on/off compromised high wattage IoT devices.

Botnets causing blackouts: how coordinated load attacks can destabilize the power grid. [ACSAC 2017]

A PANOPLY OF THREATS





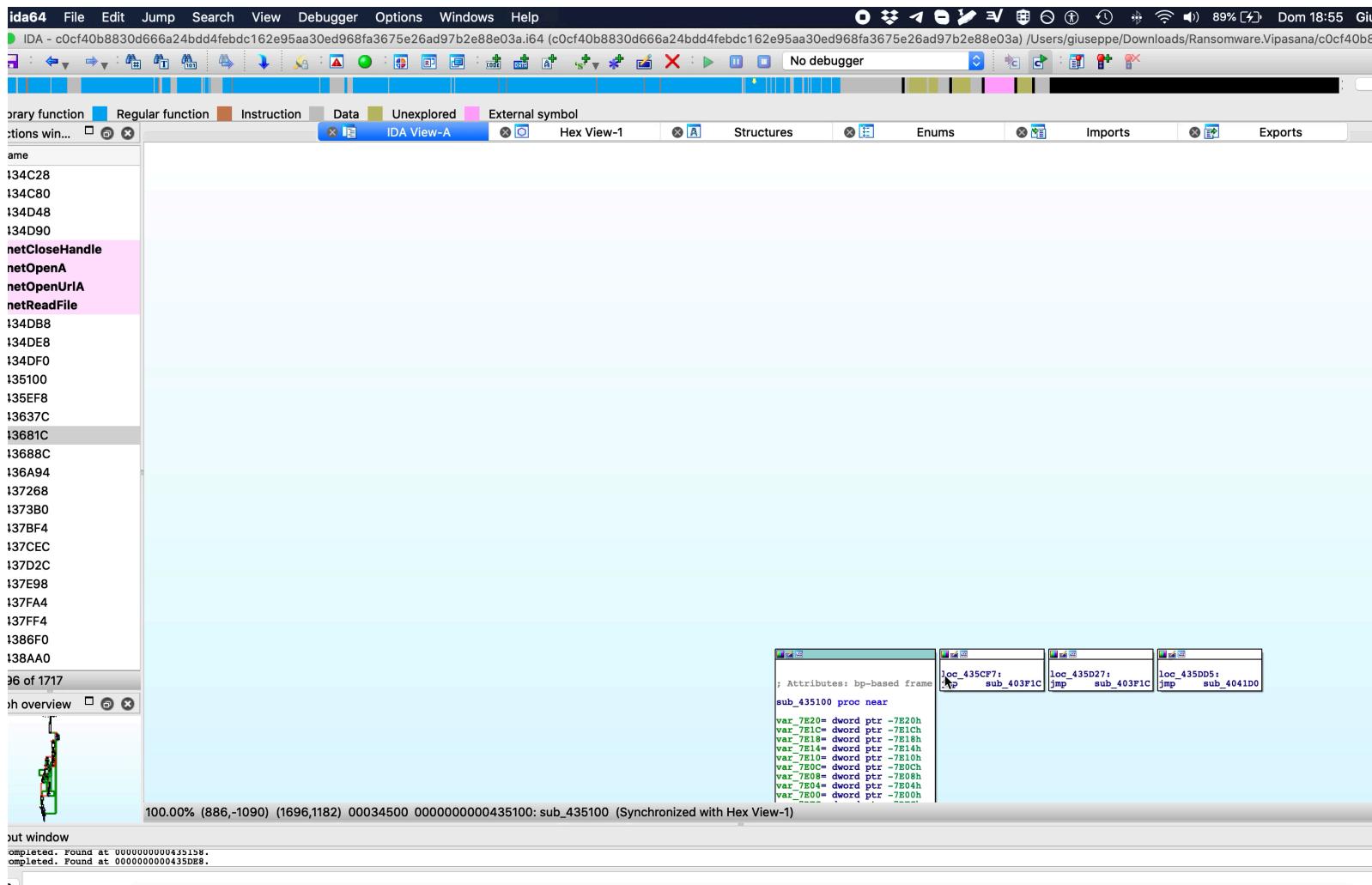
MALWARE, VULNERABILITIES AND FIRMWARES: THE HARD LIFE OF A SECURITY RESEARCHER

REVERSING A MALWARE

The screenshot shows the IDA Pro interface for reversing a 64-bit malware sample. The main window displays assembly code with several callouts highlighting specific sections:

- Function list:** On the left, a tree view lists numerous sub-functions, mostly named "sub_402XXX".
- Assembly windows:** Several assembly windows are open, showing different parts of the code:
 - loc_43AD19:** A noreturn bp-based frame containing local variables (var_5C, var_3C, var_38, var_34, var_30, var_2C, var_28, var_20, var_1C, var_18, var_14) and stack operations (push, mov).
 - loc_43A4B8:** A section starting with push eax, followed by a loop involving dec ecx and jnz short loc_43A4B8.
 - Bottom window:** The main assembly window shows the start of the program (start) at address 000398B0, containing initializations like mov eax, offset dword_43A3C8 and calls to sub_406844.
- Graph overview:** A small window on the left shows the control flow graph for the current function.
- Output window:** At the bottom, it shows the IDA Python version and copyright information.

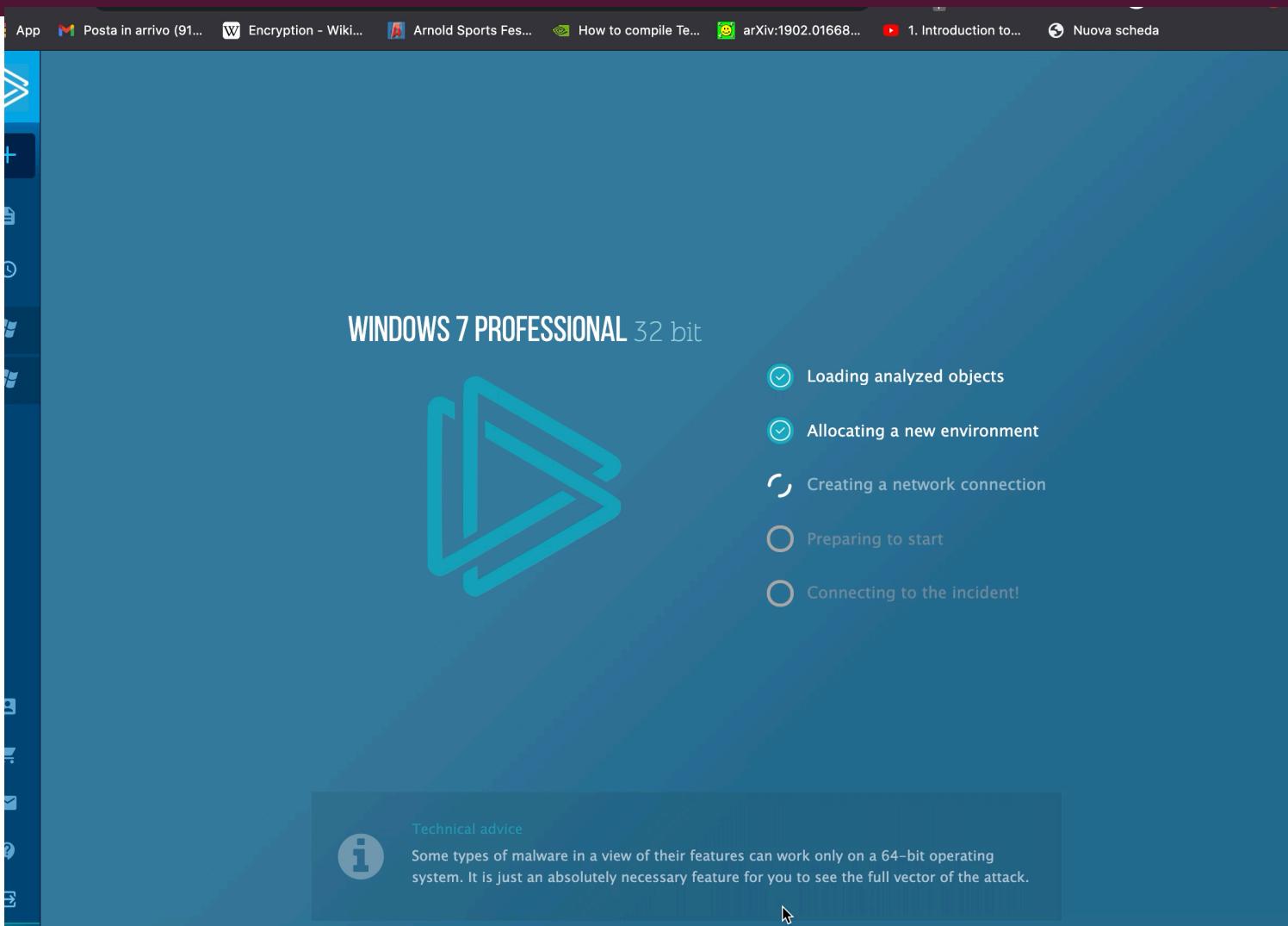
REVERSING A MALWARE



AND THAT WAS EASY....

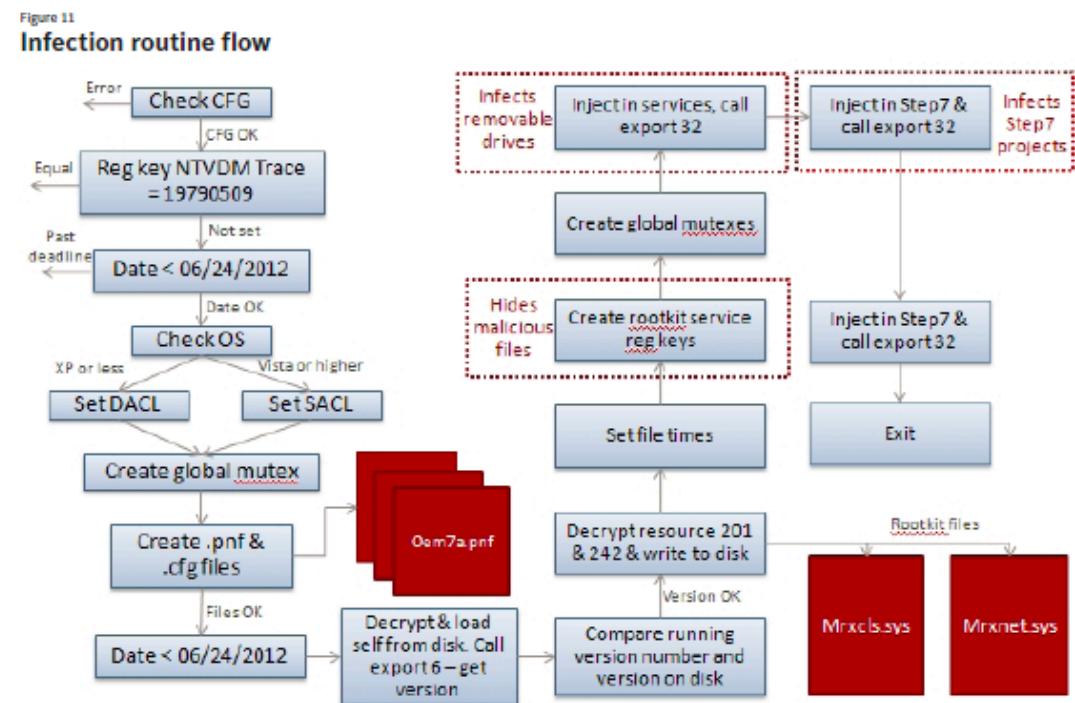
- Malware binaries are often «stripped» removing function and variable name;
- The strings inside a binary can be encrypted with a custom encryption algorithm;
- The libraries can be statically imported and can be indistinguishable from malware code;
- Packing;
- The refined ones have inside a virtual machine that runs the actual malware code.

..IN DOUBT RUN IT?



..IN DOUBT RUN IT? NOT SO SIMPLE!

A lot of malware have anti-vm techniques;
Other are tailored against a target and
show a normal behaviour;

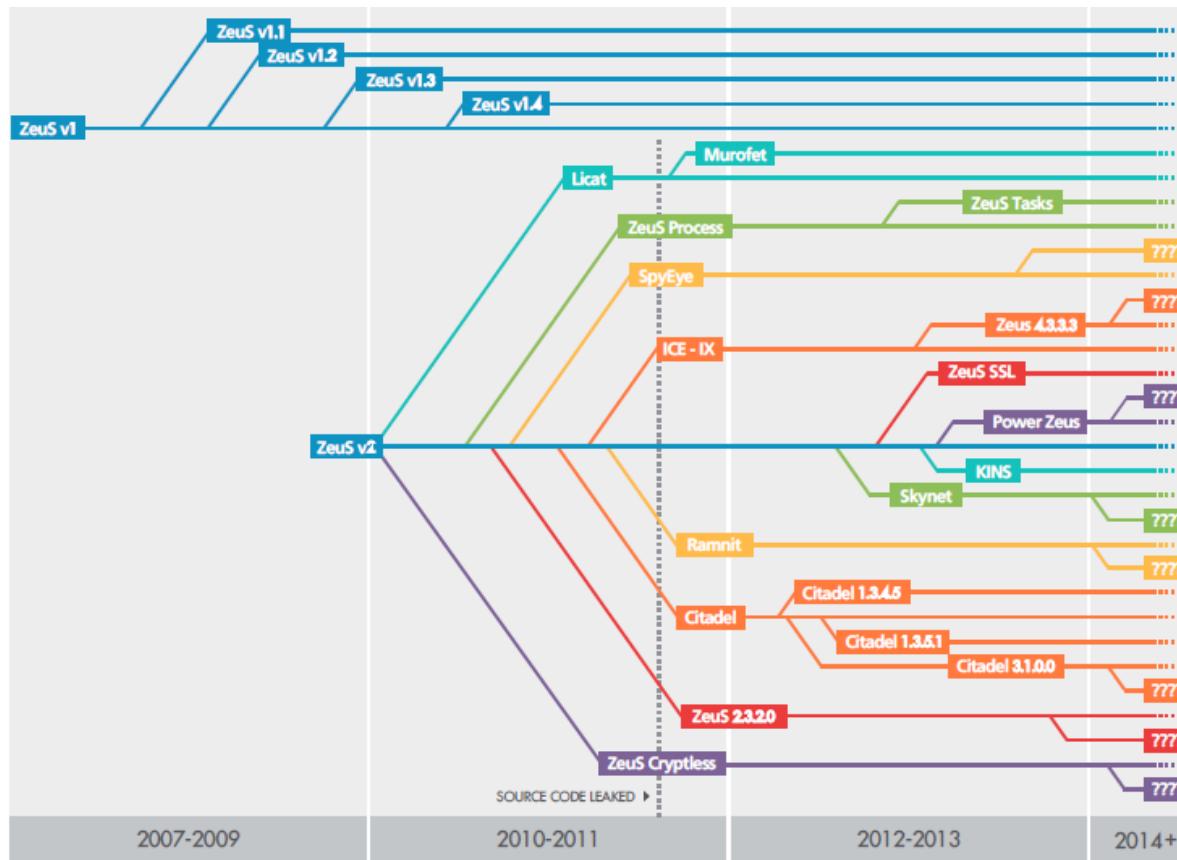


https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_WP.pdf

Understanding Linux Malware: https://reyammer.io/publications/2018_oakland_linuxmalware.pdf

REVERSING REAL-WORLD BINARIES IS A NIGHTMARE

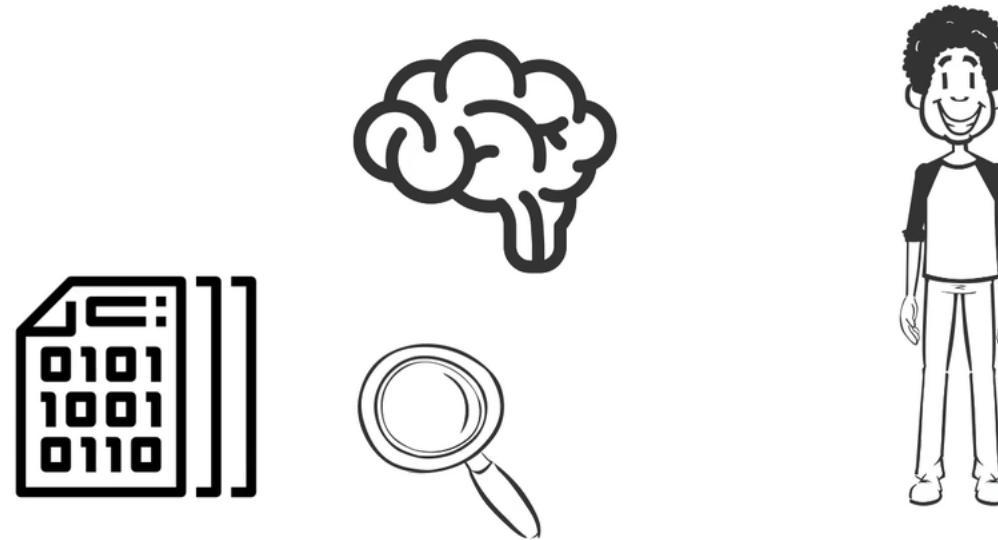
Malware developers produce variants to minimize the effort required to evade updated security defences



REVERSING REAL-WORLD BINARIES IS A NIGHTMARE

It requires skilled people and a lot, a lot, and I mean a lot of time





MACHINE LEARNING: A LIFESAVER?

MACHINE LEARNING FOR DETECTING MALWARES

Binary classification problem: Malware or Not

- Thousands of papers using thousands of techniques (DNN, Random Forest, SVM, ...) (see [0] for a recent survey);
- Static, Dynamic or Hybrid Approach;
- Extracting features: system call, functions calls [1], strings, file name, dns requests, ...
- Without manual features: MalvConv [2]



ember
an open source malware
classifier and dataset

AUC 0.99..

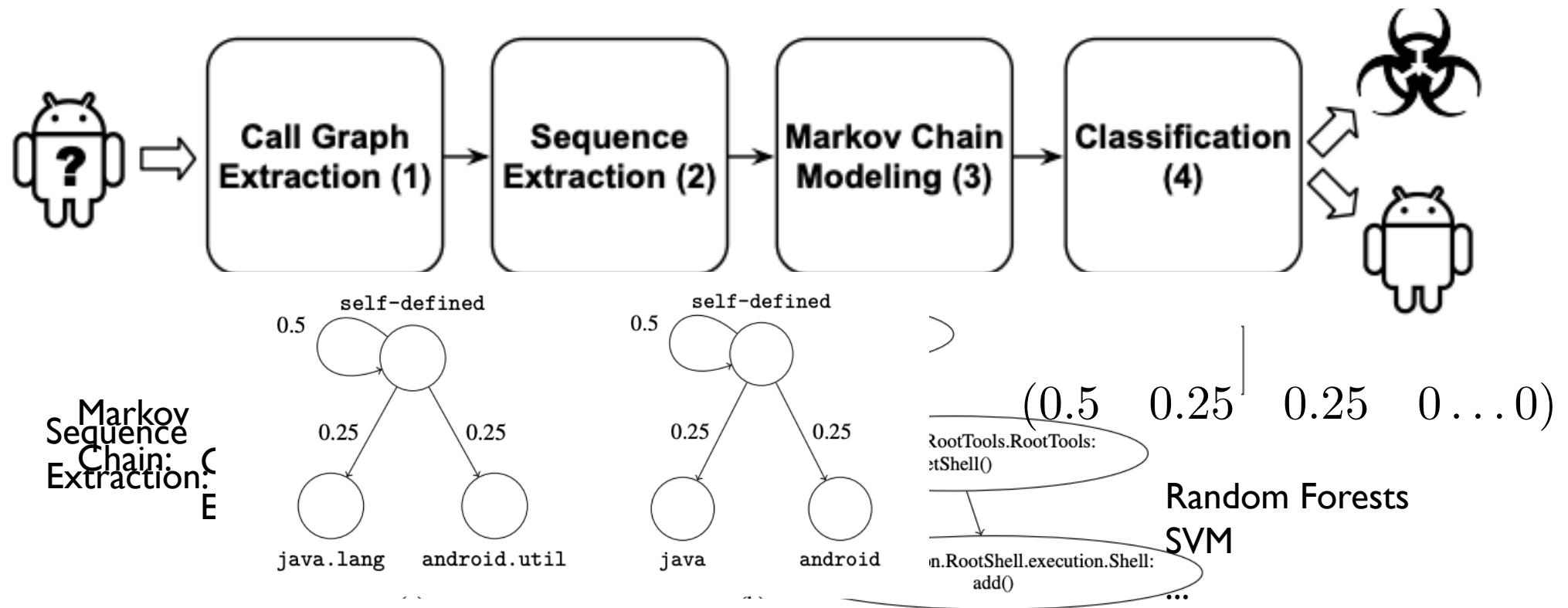
[0] Ucci et al. Survey of Machine Learning Techniques for Malware Analysis. 2018

[1] Enrico Mariconti et al. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. NDSS 2017

[2] E. Raff et al.. Malware detection by eating a whole exe. arXiv preprint arXiv:1710.09435, 2017

MACHINE LEARNING FOR DETECTING MALWARES

MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models



Images taken from:

[1] Enrico Mariconti et al. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. [NDSS 2017](#)

THE BIAS....

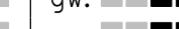
Experimental setting	Sample dates		% mw in testing set Ts							
			10% (realistic)				90% (unrealistic)			
			% mw in training set Tr				% mw in training set Tr			
	Training	Testing	ALG1 [4]		ALG2 [33]		ALG1 [4]		ALG2 [33]	
10-fold CV	gw:  mw: 	gw:  mw: 	0.91	0.56	0.83	0.32	0.94	0.98	0.85	0.97
Temporally inconsistent	gw:  mw: 	gw:  mw: 	0.76	0.42	0.49	0.21	0.86	0.93	0.54	0.95
Temporally inconsistent gw/mw windows	gw:  mw: 	gw:  mw: 	0.77	0.70	0.65	0.56	0.79	0.94	0.65	0.93
Temporally consistent (realistic)	gw:  mw: 	gw:  mw: 	0.58	0.45	0.32	0.30	0.62	0.94	0.33	0.96

Image from: [Feargus Pendlebury et al TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. USENIX Security 2019:](#)

THE BIAS....

Experiment	% mw in testing set Ts	
	10% (realistic)	90% (unrealistic)
	% mw in training set Tr	% mw in training set Tr
1	10%	90%
Temporally independent	10%	90%
Temporally independent gw/mv	10%	90%
Temporally dependent	10%	90%

Manual analysis is still necessary!

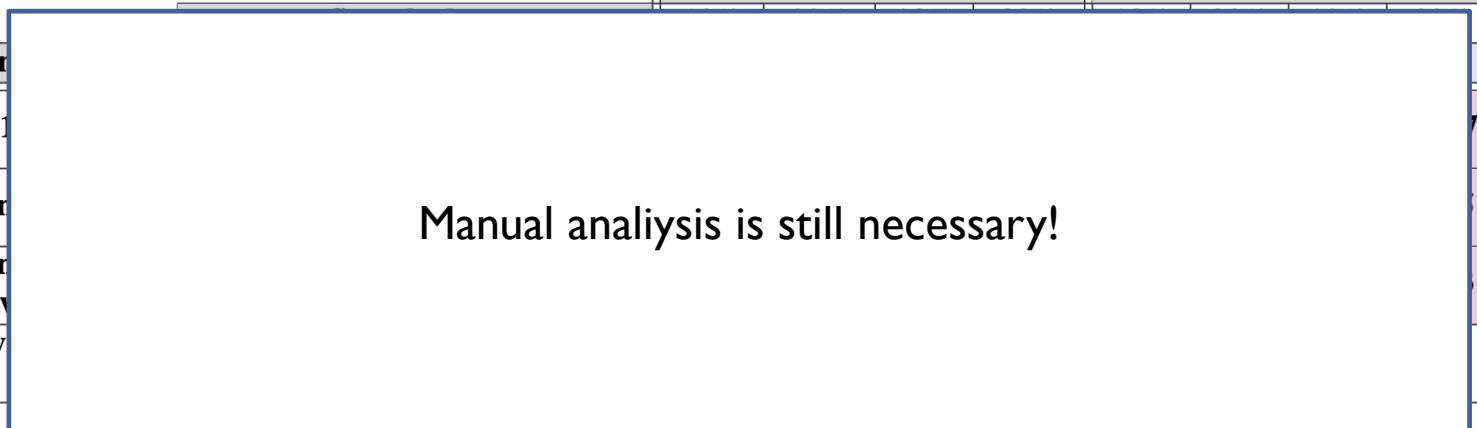


Image from: Feargus Pendlebury et al TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. USENIX Security 2019:

[1] Enrico Mariconti et al. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. NDSS 2017

AN AID TO REVERSE ENGINEERING

Machine learning has been used in a successfull way to help manual analysis:

- Function boundaries
- Binary Similarity
- Function Naming
- Decompiling
- ...

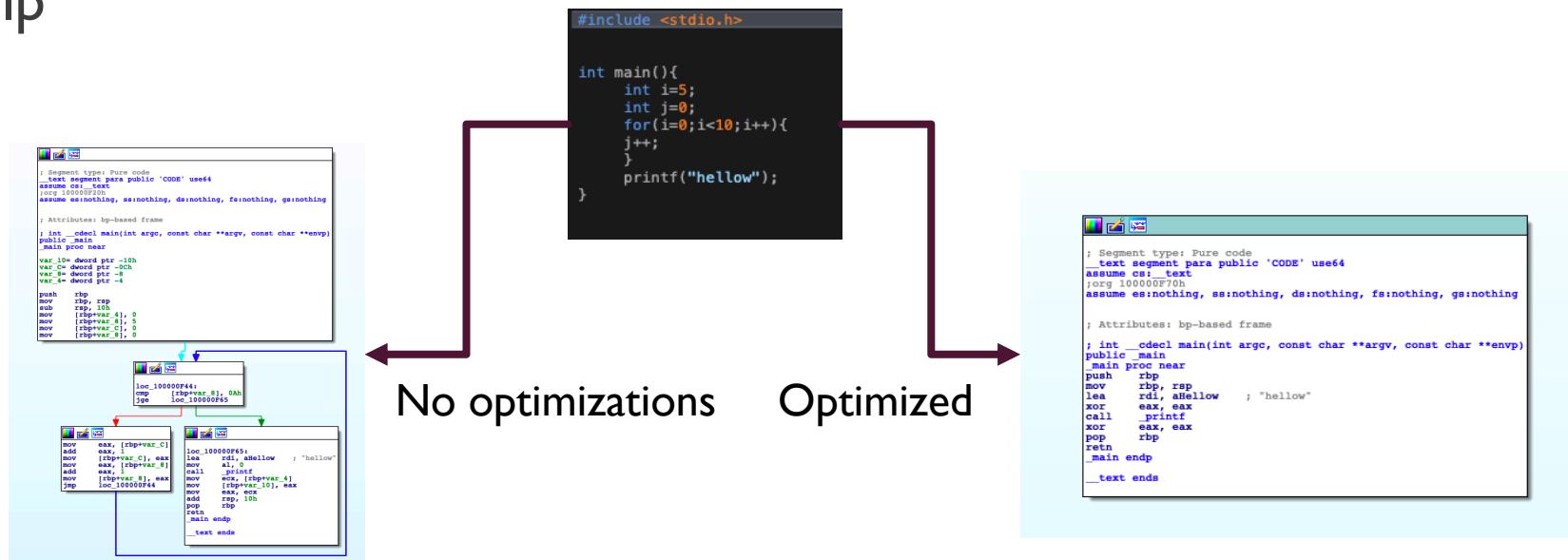
The screenshot shows a terminal window titled "safe_apt_classification" running on a Mac OS X system. The command "cd presentazione" and "ls" are run to show files: a.c, a.out.dSYM, gonnacry.asm.txt, unopt, a.o, a.o.dSYM, mirai_asm.txt, unopt.dSYM, and opt. The command "r2 a.o" is then run, followed by the message "-- You see it, you fix it!" and the address "[0x100000f90]>". Below the terminal, the assembly code for the program is displayed, showing instructions like "mov eax, [rbp*1+-28]", "cdqe", "lea rdx, [rax*1+-1]", and "add rax, rdx". The assembly code is heavily annotated with labels such as "X_jge_HIMM", "X_mov_eax", "X_cdqe", "X_lea_rdx", "X_mov_rax", "X_add_rax", "X_m", "X_jne", "X_m", "X_0x", "X_mo", "X_1+", "X_af", and "X_ef". The right side of the interface shows memory dump sections for addresses 176, -16, and -4.

```
safe_apt_classification
[giuseppe@Giuseppes-MacBook-Pro-2 ~ % cd presentazione
[giuseppe@Giuseppes-MacBook-Pro-2 presentazione % ls
a.c          a.out.dSYM      opt.dSYM
a.o          gonnacry.asm.txt  unopt
a.o.dSYM    mirai_asm.txt   unopt.dSYM
a.out        opt
[giuseppe@Giuseppes-MacBook-Pro-2 presentazione % r2 a.o
-o -- You see it, you fix it!
ea [0x100000f90]>
0x
X_
bp
l_
'
X_
bp
*1
_c
IM
X_
6]
_m
IM
[_
jn
_m
X_
0x
l_
mo
*1
X_
X
1+
ax,_[rbp*1+-28] X_jge_HIMM X_mov_eax,_[rbp*1+-8] X_cdqe X_lea_rdx,_[rax*1+-1] X_mov_rax,_[rbp*1+-24] X_add_rax,_rdx X_m
+8],_0x0 X_jne_HIMM X_mov_rax,_[rbp*1+-24] X_jmp_HIMM X_mov_eax,_0x0 X_leave X_ret
X_push rbp X_mov rbp _rsp X_mov eax edi X_mov [rbp*1+-4], al X_cmp [rbp*1+-4],_0x40 X_le HMM X_cmp [rbp*1+-4],_0x5a
```

AN AID TO REVERSE ENGINEERING

Machine learning has been used in a successfull way to help manual analysis:

- Function boundaries
 - **Binary Similarity**
 - Function Naming
 - Decompiling
 - ...



A REVOLUTION IN SOFTWARE ENGINEERING

The BigCode revolution (see **A Survey of Machine Learning for Big Code and Naturalness – [Allamanis et al. 2018]**):

- Use the big codebase given by opensource projects.

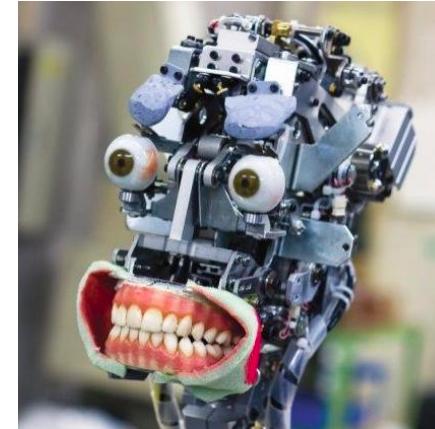
The naturalness hypothesis: «*Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools*».

PECULIARITIES OF SOURCE CODE

Code is a mean of communication between humans and machines (*bimodal*)



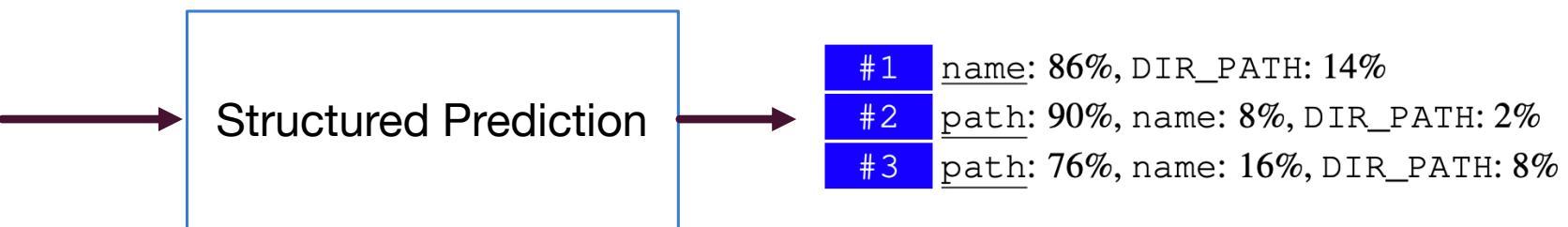
```
author min
This class will use the ghost
to insert a collection of particles which
have no net charge. This is used to calculate
chemical potential and activity coefficient
ass widom : public analysis {
private:
    average<double> expsum; //!< Average of 1
protected:
    int ghostin;
    long long int cnt;      //!< Count test
    vector<particle> g;     //!< List of ghost
public:
    widom(int n=10);
    string info();
    void add(particle);
    void add(container &);
    void insert(container &, energybase &,
    void check(checkValue &);
```



- Code is *executable*
- Code is *formal*

APPS – GENERATIVE MODELS

```
var path = CreateFileName( #1 );
bitmap.Save( #2 , ImageFormat.Png);
return #3;
```



LEARNING TO REPRESENT PROGRAMS WITH GRAPHS [Allamanis et al. 2018]

APPS – GENERATIVE MODELS

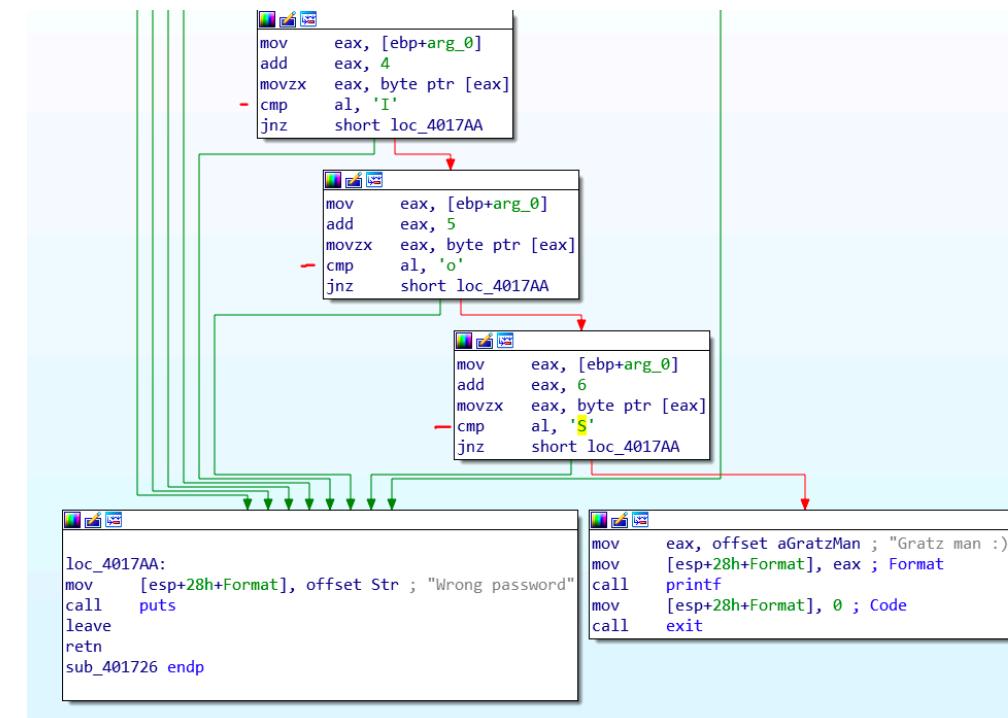
Structured Prediction

```
public ArraySegment<byte> ReadBytes(int length) {  
    int size = Math.Min(length, _len - _pos);  
    var buffer = EnsureTempBuffer(length);  
    var used = Read(buffer, 0, size);
```

...WHAT ABOUT MACHINE CODE?

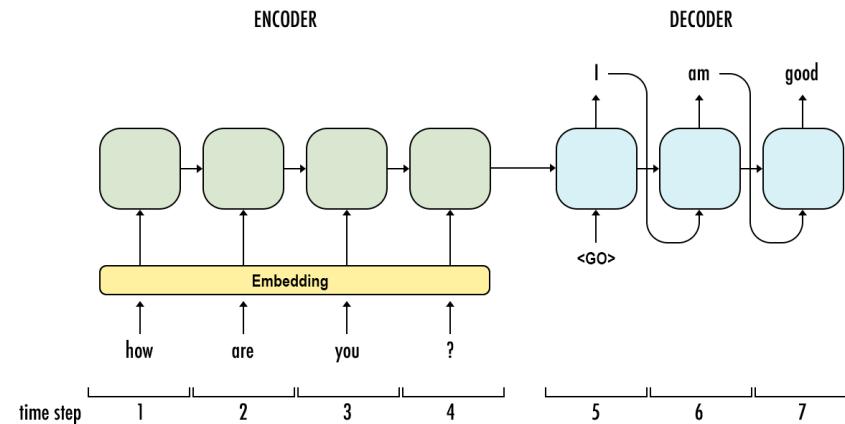
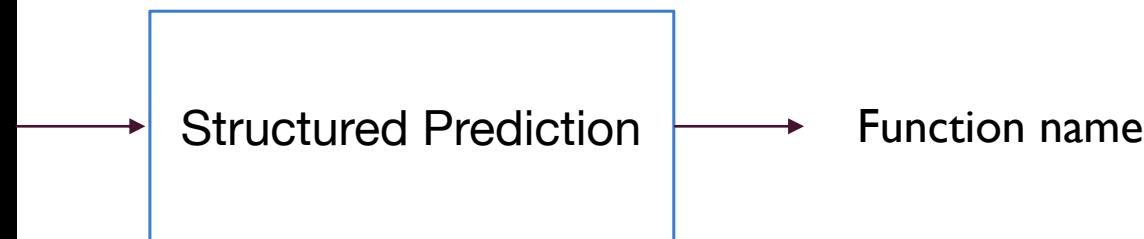
Machine code is harder:

- No variable names
- No types
- Really long sequences of instructions
- No function names (if stripped)



FUNCTION NAMING

```
80483b4: 55          push %ebp
80483b5: 89 e5        mov %esp,%ebp
80483b7: 83 e4 f0     and $0xffffffff0,%esp
80483ba: 83 ec 20     sub $0x20,%esp
80483bd: c7 44 24 1c 00 00 00 00  movl $0x0,0x1c(%esp)
80483c4: 00
80483c5: eb 11        jmp 80483d8 <main+0x24>
80483c7: c7 04 24 b0 84 04 08  movl $0x80484b0,(%esp)
80483ce: e8 1d ff ff   call 80482f0 <puts@plt>
80483d3: 83 44 24 1c 01  addl $0x1,0x1c(%esp)
80483d8: 83 7c 24 1c 09  cmpl $0x9,0x1c(%esp)
80483dd: 7e e8        jle 80483c7 <main+0x13>
80483df: b8 00 00 00 00  mov $0x0,%eax
80483e4: c9           leave
80483e5: c3           ret
80483e6: 90           nop
80483e7: 90           nop
80483e8: 90           nop
80483e9: 90           nop
80483ea: 90           nop
```



FUNCTION NAMING - PERFORMANCES

Real Name	Predicted Name
client write transition	server write transition
file seek shared	file get size
aes encrypt	shall block enc
section write packet	build array
drop transform by id	remove by id
read array binary	array get slice
generate key ex	digest verify final

Predictions on normal file

Real Name	Predicted Name
attack_kill_all	kill all
attack_get_opt_ip	addr
has_exe_access	read pid file
util_memsearch	find
table_retrieve_val	get
util_strncmp	compare
mem_exists	str cmp

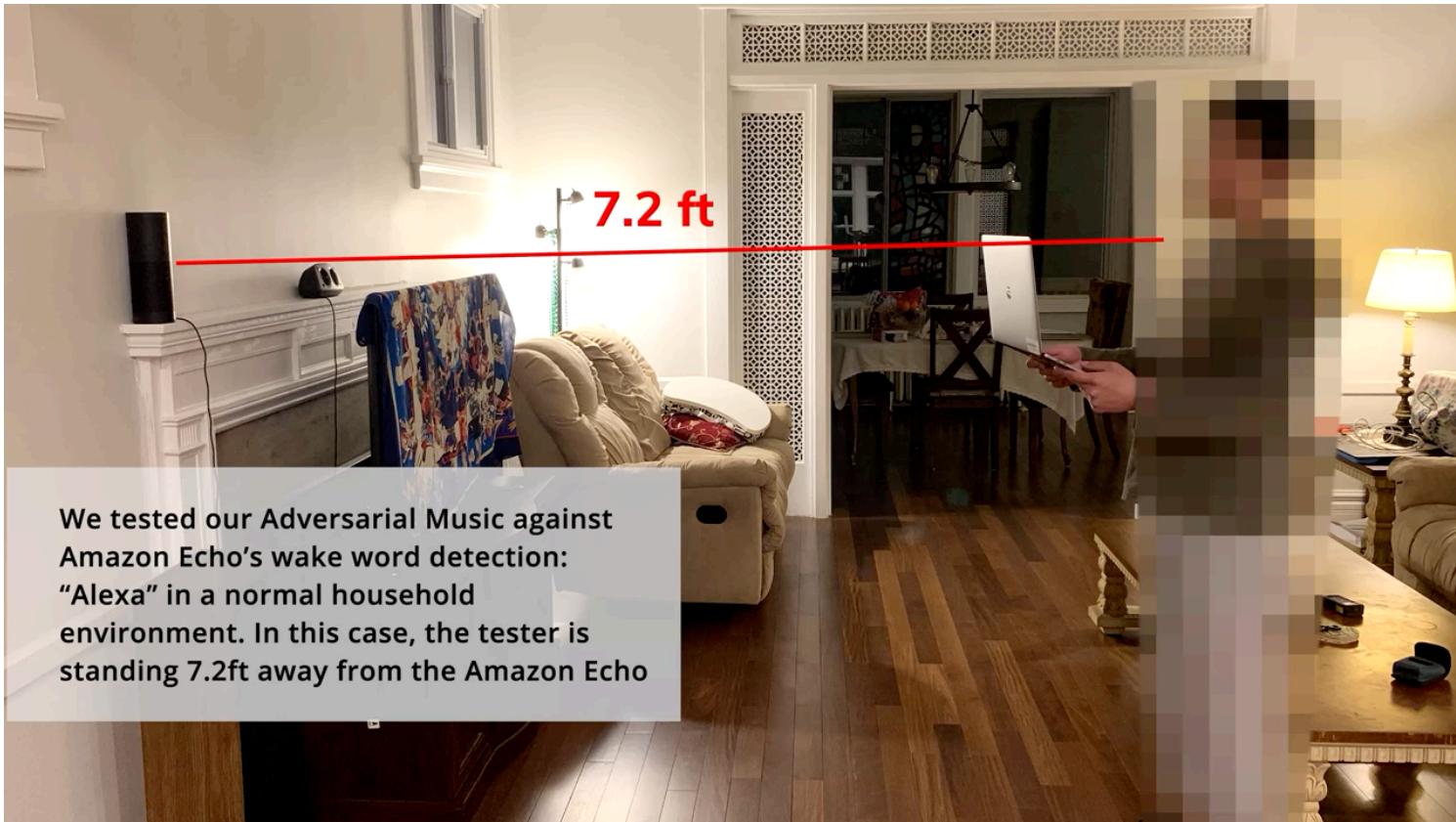
Predictions on mirai



...OR IS THE NEXT THREAT?

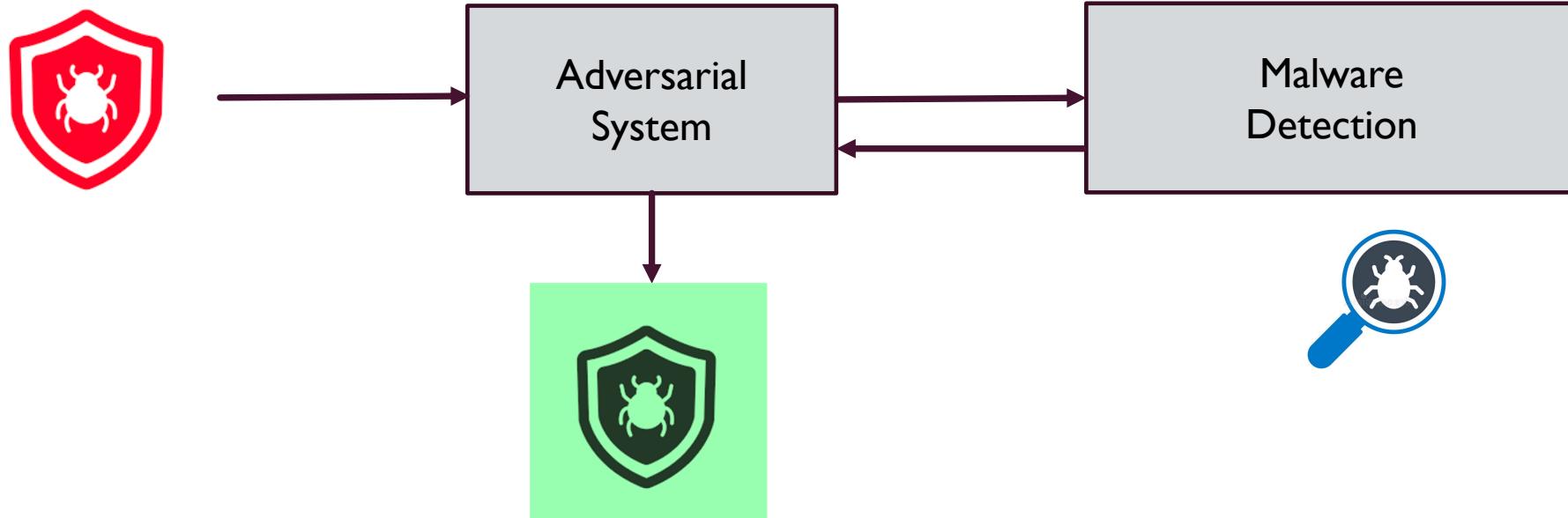


ATTACKS ON ML SYSTEMS



Adversarial Music: Real World Audio Adversary Against Wake-word Detection System: <https://arxiv.org/pdf/1911.00126.pdf>

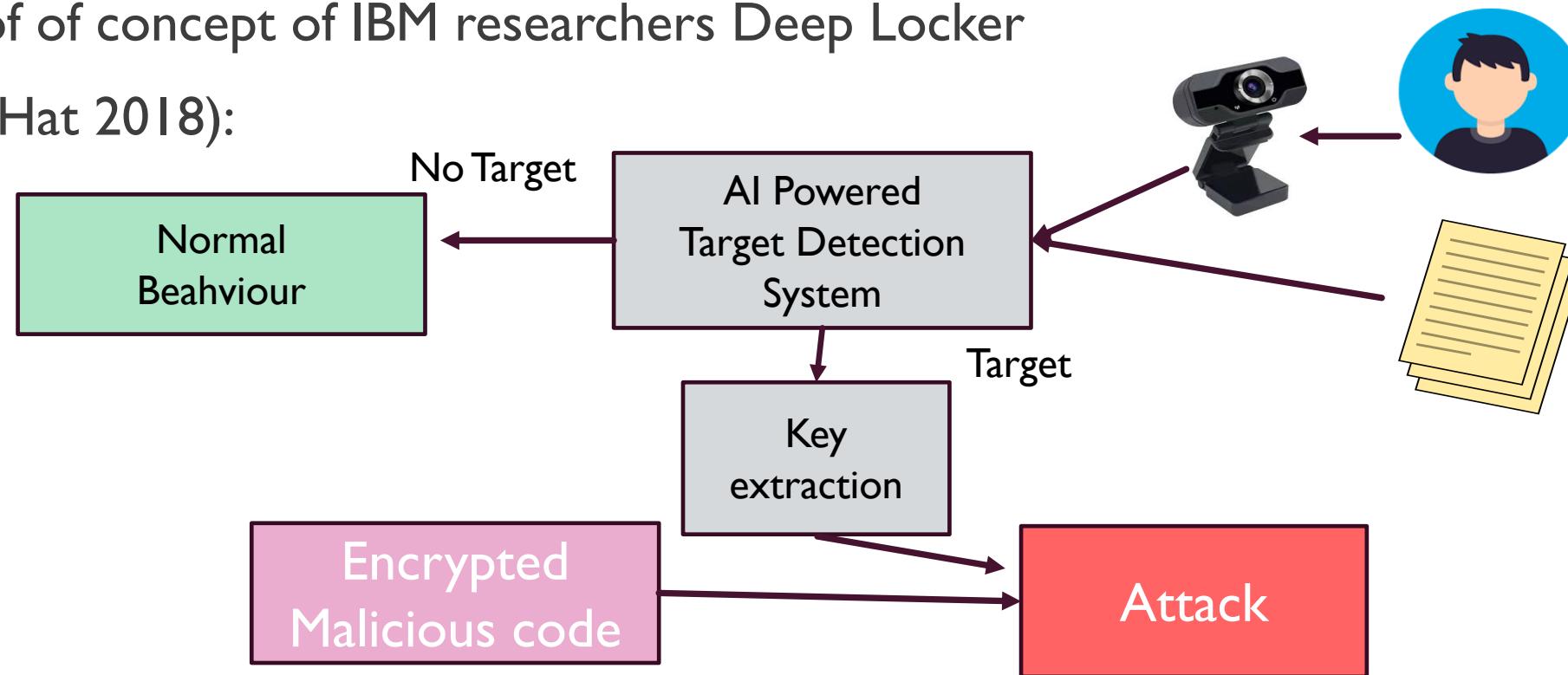
CREATING EVASIVE MALWARE



- Anderson et al. Learning to Evoke Static PE Machine Learning Malware Models via Reinforcement Learning -
<https://arxiv.org/pdf/1801.08917.pdf>
- Hu et al. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN -
<https://github.com/yanminglai/Malware-GAN>

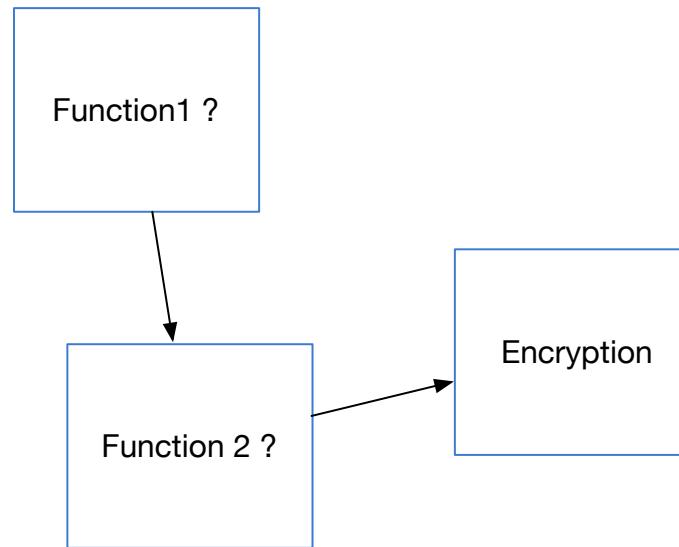
AI POWERED MALWARE

A proof of concept of IBM researchers Deep Locker
(Black Hat 2018):



HOMEWORK – BINARY FUNCTIONS CLASSIFICATION

MOTIVATION



- During reversing of an unkown software it is often useful to find specific functions
- As an example encryption functions in a malware

OBJECTIVE

- You have to classify binary functions belonging to 4 classes:
 - Encryption
 - String Manipulation
 - Math
 - Sorting

DATASET LINK:

<https://drive.google.com/file/d/1vKT9OzJwM6gKACyElzECW49X5nFeVxad/view?usp=sharing>

DATASET

- You are given a dataset of 14397 functions, each function belongs to one of the aforementioned classes (Encryption, String Manipulation, Math, Sorting)
 - 2724 are Encryption Functions (label: Encryption)
 - 3104 are String Manipulation (label: String)
 - 4064 are Sorting Functions (label: Sort)
 - 4504 are Math Functions (label: Math)
- You are given a blind test file (functions without labels). You have to deliver a simple text file with your predictions for the blind test (keep the original order!).

DATA FORMAT

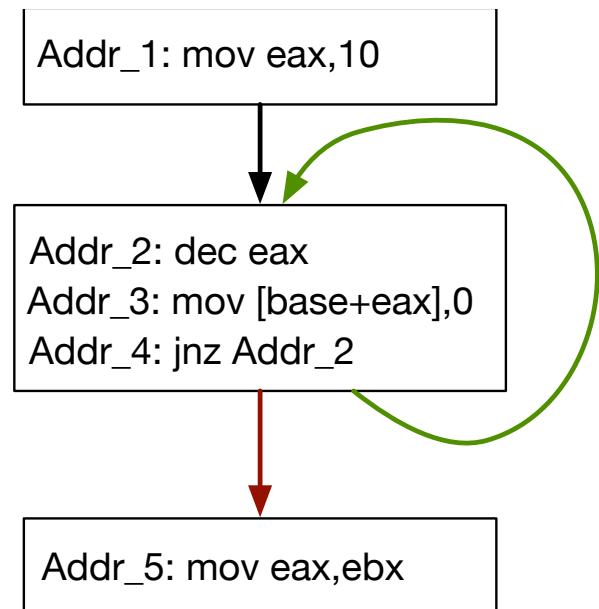
- Each file is a JSONL file (each line is a json object).
- Each item is a dictionary {key1: value1, key2: value2,...}. The keys are:
 - ID: unique id of each function
 - semantic: the label of each function in {math, sort, encryption, string}
 - lista_asm: the linear list of assembly instruction of each function
 - cfg: the control flow graph encoded as a networkx graph

DATA FORMAT

```
Addr_1: mov eax,10  
Addr_2: dec eax  
Addr_3: mov [base+eax],0  
Addr_4: jnz Addr_2  
Addr_5: mov eax,ebx
```

- list_a_asm: the linear list of assembly instruction of each function
- cfg: the control flow graph encoded as a networkx graph

ach line is a json object).
key1: value1, key2: value2,...}. The keys are:
nction in {math, sort, encryption, string}



POSSIBLE FEATURES

We have to think about reasonable features using the fact that we are experienced developers.

Encryption ?

Fairly Complex Functions
(think about AES):

1. A lot of nested For and IF
-> Complex CFG
2. They use xor a lot (and I mean a lot..), shifts and bitwise operations
3. Extremely long

Sorting ?

- Usually the logic is simple:
1. One or two nested For and some helper functions
 2. They use compare and moves operations

Math ?

Not surprisingly they use a lot of arithmetic operations, and sometimes they do operations on vectors and matrices:

1. They usually use floating point instructions, special registers **xmm***

String Manipulation ?

A lot of comparisons and swap of memory locations

POSSIBLE FEATURES

We have
developed

Encryption

Fairly Complex F
(think about AES)

1. A lot of nested loops
-> Complex
2. They use xor a lot
mean a lot..)
bitwise operations
3. Extremely long

Good features could be:

- 1) Features that captures the complexity of the CFG (e.g., number of loops, edges, cyclomatic complexity, ...)
- 2) Features that take into account the kind of instructions contained in each function:
 - 1) You can group instructions into broad categories (bitwise operation, floating point, data move, etc...)
- 3) Global Features:
 - 1) Number of external calls
 - 2) Number of instructions
 - 3) Usage of registers (that you can also group in categories)

enced
pulation ?

sons and swap
ons

REFERENCES

https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

<http://web.cecs.pdx.edu/~apt/cs491/x86-64.pdf>

https://networkx.org/documentation/stable/reference/readwrite/json_graph.html