COMPUTER SCIENCE
& ENGINEERING
TEXAS A&M UNIVERSITY

JSTAP: Malicious JavaScript Detection

Dr. Martin "Doc" Carlisle

Premise

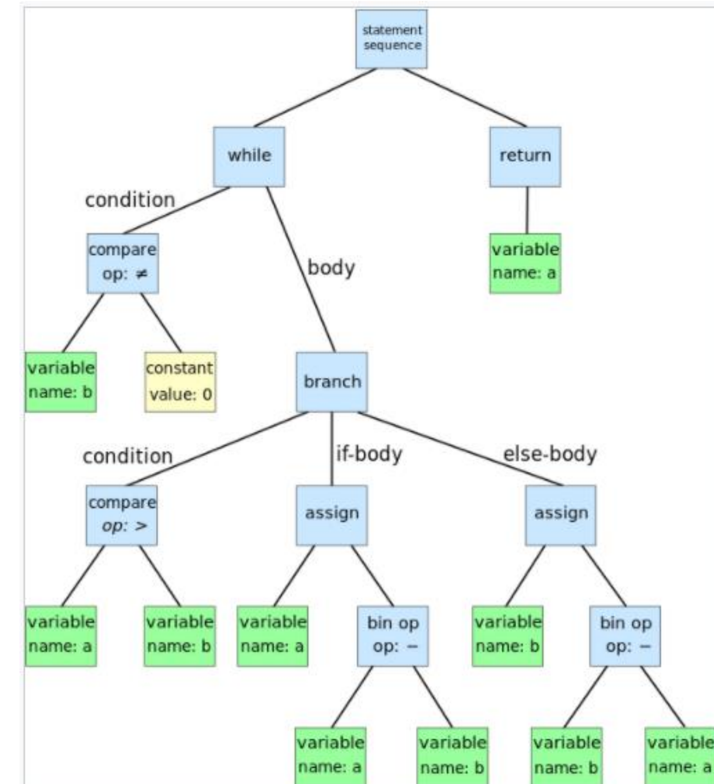
- Find malicious JavaScript
  - Bitcoin mining
  - Abuse browser vulnerabilities
  - Perform static analysis with abstract syntax trees and random forests
    - Static analysis means we don’t run the code at all

Static Analyses (I)

- Abstract Syntax Tree
  - Derived from grammar of programming language

An abstract syntax tree for the following code for the Euclidean algorithm:

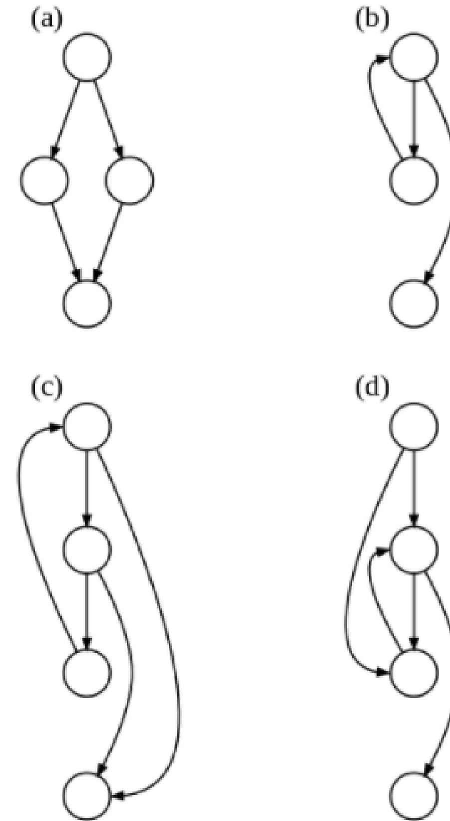
```
while b ≠ 0
  if a > b
    a := a − b
  else
    b := b − a
return a
```



Static Analyses (II)

- Control Flow Graph
  - Shows program flow (calls, selection, loops)

Some CFG examples:
(a) an if-then-else
(b) a while loop
(c) a natural loop with two exits, e.g. while with an if...break in the middle; non-structured but reducible
(d) an irreducible CFG: a loop with two entry points, e.g. goto into a while or for loop



Static Analyses (III)

- Program Dependence Graph
  - Includes data and control dependencies
    - A=B*C
    - D=A*E+1 (this depends on the prior statement)

    - if (A) then
      - B=C*D (this depends on value of A)
    - endif

JavaScript tokens

```
1 x.if = 1;
2 var y = 1;
3 if (x.if == 1) {d = y;}
```

Listing 1: JavaScript code example

Table 1: Lexical units extracted from the code of Listing 1

Token	Value	Token	Value	Token	Value
Identifier	x	Numeric	1	Punctuator	)
Punctuator	.	Punctuator	;	Punctuator	{
Keyword	if	Keyword	if	Identifier	d
Punctuator	=	Punctuator	(	Punctuator	=
Numeric	1	Identifier	x	Identifier	y
Punctuator	;	Punctuator	.	Punctuator	;
Keyword	var	Keyword	if	Punctuator	}
Identifier	y	Punctuator	==		
Punctuator	=	Numeric	1		

JavaScript example

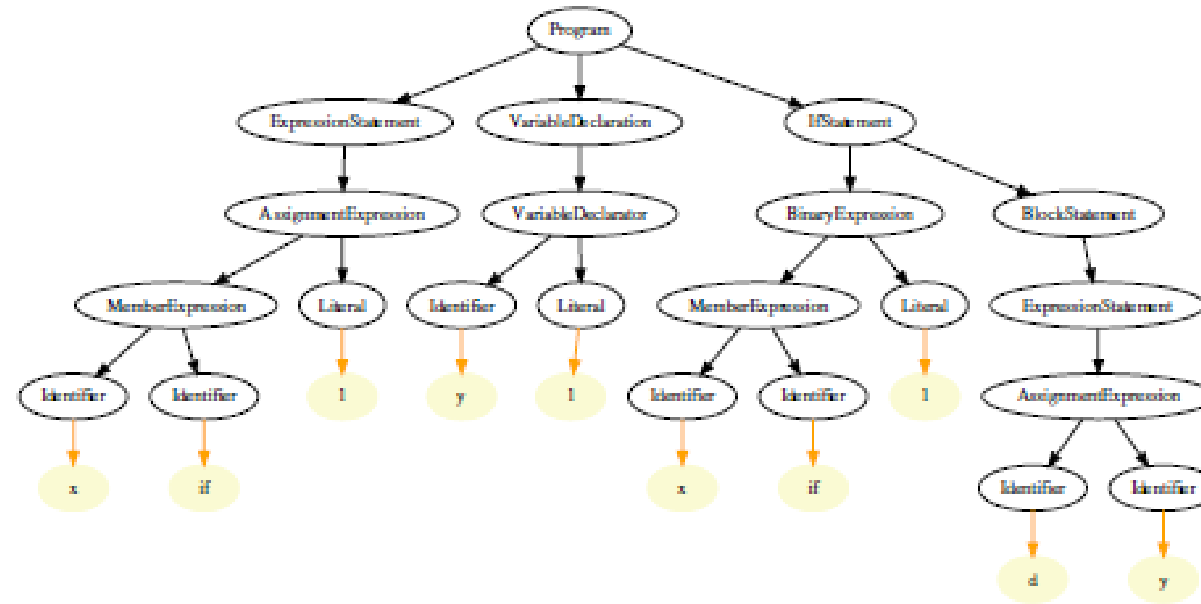


Figure 2: AST corresponding to the code of Listing 1

```
1  x.if = 1;
2  var y = 1;
3  if (x.if == 1) {d = y;}
```

Listing 1: JavaScript code example

JavaScript example

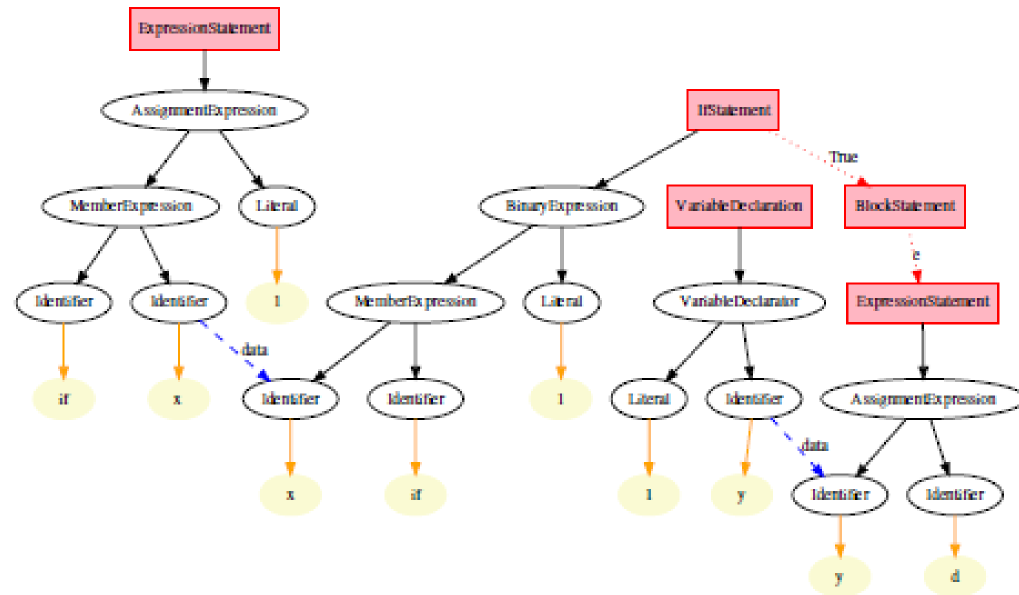


Figure 3: AST of Listing 1 extended with control flow (red dotted edges) and data flow (blue dashed edges)

```
1 x.if = 1;
2 var y = 1;
3 if (x.if == 1) {d = y;}
```

Listing 1: JavaScript code example

N-grams

- Simple way to analyze token sequences

- Example with n=3

```
ID = ID + NUM      ⟶  { (ID = ID), (= ID +), (ID + NUM) },
SET a.b to "x"     ⟶  { (SET a.b to), (a.b to "x") }.
```

JSTAP n-grams (I)

- Depth-first pre-order traversal of AST
- For CFG, also traverse AST, but only nodes linked by control flow edge.
  - Traverse sub-AST for each node with control flow once
- Similar for PDG, considering data flow
- Independent n-grams for tokens, AST, CFG, PDG-Data Flow and PDG-Control Flow

JSTAP n-grams (II)

- Set n=4 (experimentally)
- Use chi-squared test to check for correlation, keep $\chi^2 \geq 6.63$ (confidence of 99%)
  - Lets us throw away a lot of n-grams

Table 2: Number of relevant features per module

	Tokens	AST	CFG	PDG-DFG	PDG
ngrams	602	11,050	18,105	17,997	24,706
value	24,912	45,159	36,961	45,566	46,375

JSTAP dataset

- 131,448 malicious JavaScript files
  - German Federal Office for Info Security
  - Hynek, DNC, GeeksOnSecurity, Virus Total
- 141,768 benign files
  - Top 10,000 Tranco websites
  - JS from Exchange 2016 and Team Foundation Server 2017
    - So obfuscation isn't confused with maliciousness

JSTAP Classifier Training

- Select 10,000 malicious and benign randomly for training
  - Additional 5,000 of each for validation
- Repeat 5 times and average detection results

An interesting claim

- Fass et al. “For this reason, AUC and F-measure would be heavily biased by the composition of our test sets”
- Fawsett “ROC curves have an attractive property: they are insensitive to changes in class distribution. If the proportion of positive to negative instances changes in a test set, the ROC curves will not change.”

JSTAP Results

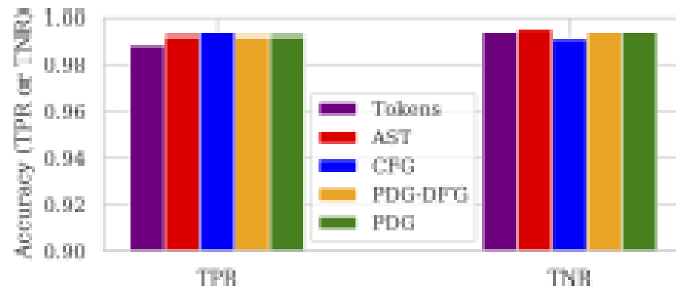


Figure 4: Accuracy comparison with the ngrams approach

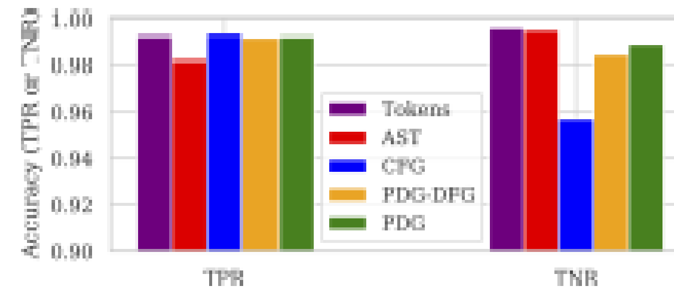


Figure 5: Accuracy comparison with the value approach

JSTAP vs others

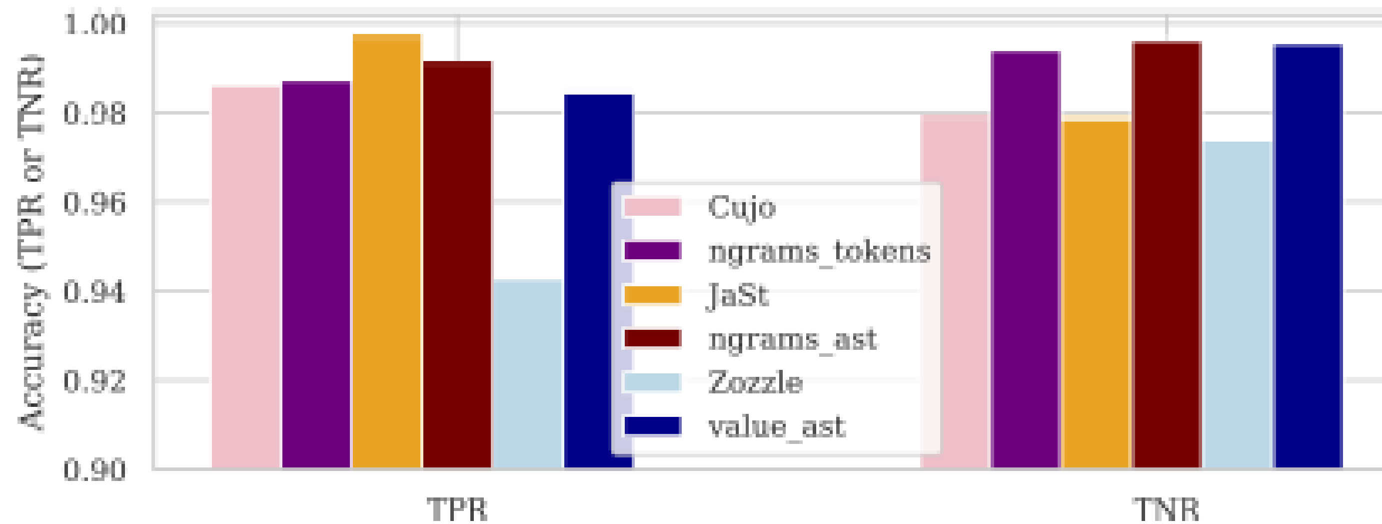


Figure 6: Accuracy comparison between related work and our improved corresponding implementations

Cujo: 4-grams better than 3-grams, and random forest better than SVM
Zozzle: all nodes (not just exprs and var decls), random forest vs naïve Bayes
JAST: do not simplify but use χ^2 test to reduce size of feature space

JSTAP results

- Two step process
- First phase
  - Unanimous voting, classifies 93% of data with 99.73% accuracy
- Second phase
  - Unanimous voting, classifies 6.5% of data with accuracy still over 99%

Evasion techniques

- Add more benign features
- Copy malicious into larger benign file

COMPUTER SCIENCE
& ENGINEERING
TEXAS A&M UNIVERSITY

Malware Detection by Extreme Abstraction

Dr. Martin "Doc" Carlisle

Premise

- Find malicious Windows EXEs by abstract execution
  - Less precise than virtualization or emulation

Why dynamic analysis

- Malware writers deliberately obfuscate to defeat static tools
  - Example: GozNym runs trivial infinite loop in thread, then suspends thread and overwrites code with jump to previously dead code

```
0x100178c:  eb fe           jmp 0x100178c
0x100178e:  fe              --
0x100178f:  ff ff           --
0x1001791:  6a 5c           push 0x5c
```

(a) A trivial infinite loop

```
0x100178c:  b8 00 00 7e 04  mov eax, 0x47e0000
0x1001791:  ff e0           jmp eax
```

(b) Loop after being overwritten

Figure 1: Obfuscation by overwriting seven bytes

Dynamic Analysis pitfalls

- Easy to detect you are in a debugger, VM, or running Anti-virus
  - Query registry
  - IsDebuggerPresent
  - VM specific instructions
- Do long delay in hopes simulator will give up and go away

```
0x4017c0:   mov esi, dword ptr [ebp-0x26]
0x4017c3:   mov esi, dword ptr [esi]
0x4017c5:   xor esi, edi
0x4017c7:   inc edi
0x4017c8:   cmp esi, 0x90909090
0x4017ce:   jne 0x4017c0
```

Figure 2: A long delay loop

Extremely Abstract OS

- Over-approximation has more behaviors than system S, under-approximation has fewer
  - If over-approximation does no evil, great!
  - If under-approximation does, then boo!

Extremely Abstract OS

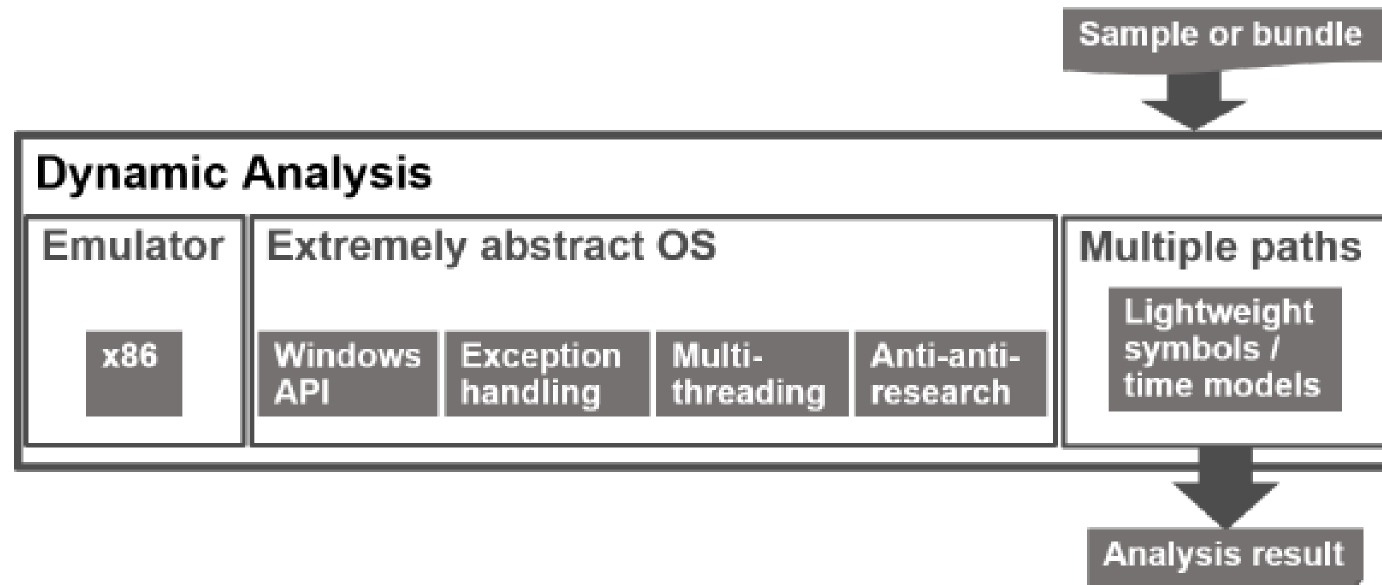


Figure 4: An extremely abstract operating system

TAMALES features

- X86 emulator
- Abstract Windows
  - Most routines return random result and ignore params
    - Over 100,000 API calls from 150 DLLs
    - Strcpy, memmove work as expected
    - Some file read/write and registry read/write
    - Network is abstract
  - Rdtsc – time-stamp counter handled specially
  - Cpuid – handled specially
- Runs on Linux (just in case….)

Another unique case

- SetErrorMode

```
x = y;
```

can be implemented (directly by the malware writer or more probably by an obfuscating compiler) as follows:

```
SetErrorMode(y);
x = SetErrorMode(arbitrary_value);
```

- Since malware writers do this, must implement for real

More malware functions

- WriteProcessMemory
- CreateRemoteThread
- NtQueueApcThread
- NtMapViewOfSection

(for code injection)

And more special cases

- Return value of 0 is success
- Esoteric API called with bad params then checking error code (they have to chase down each individually, so a path to thwart)

Multiple Paths

- Typically, use symbolic execution
  - SAT solver finds values needed to explore paths
  - Expensive, path explosion
- TAMALES just takes both paths
  - Explodes really bad

Preventing Explosions

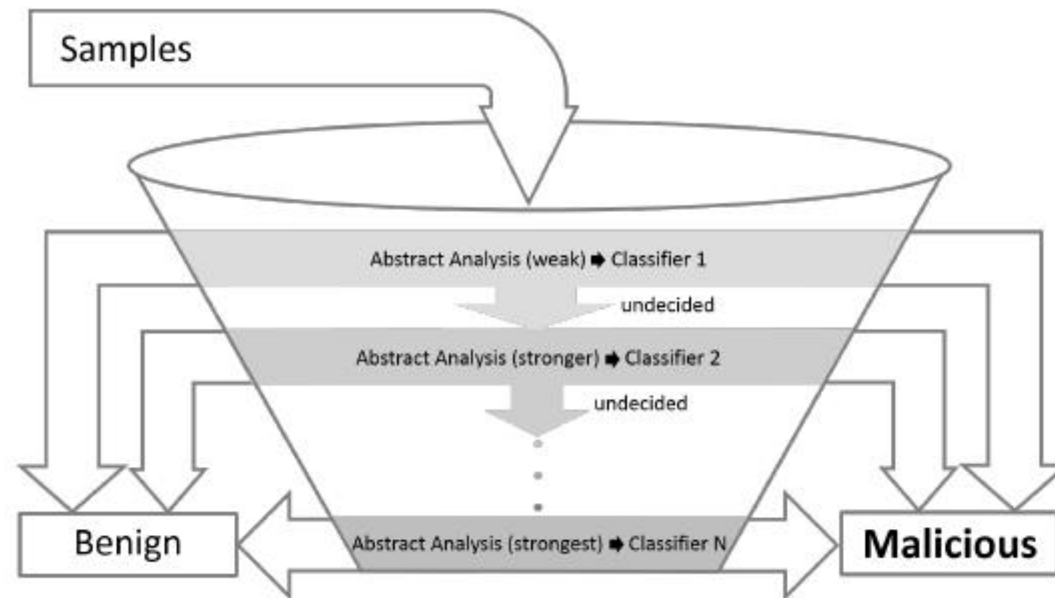


Figure 5: Architecture of TAMALES

Add more paths at each layer. Two thresholds, one to say benign, one malicious

Feature Extraction for ML

- Entropy of code/data sections
- Discrepancy between checksum header and PE
- Imported functions
- Count of API functions and x86 instructions
- Count of exceptions and types, network connections, strings
- Ratio of API functions imported to called and static vs dynamic strings

More features

- Suspicious
  - Checking for debugger
  - Obfuscation
  - Jumping into middle of API
  - Overwriting header or part of API function
  - Directly accessing OS structures
  - Creating an intentionally infinite loop

Yet more features

- Almost certainly bad
  - Malicious URLs
  - Encrypting/deleting files not created by sample
  - Overwriting Windows DLLs

More n-grams!

- 1-, 2-, 3-, and 4-grams of API calls
- 1-grams of x86 instructions
- 1-, 3-, and 6-grams of informative, suspicious, malicious features

And numeric stuff

- # of internet access attempts
- Number of unique URLs
- Connection attempts with non-standard ports
- Reputation score of target host

Data Cleaning

- Remove features that are always the same
- Scale all to [0,1]
- Feature select using information gain

- Yields 3500 features

Random Forest Classifier

- 1,600 decision trees
  - Max depth 150
  - Up to 200 features per split

Table 1: PE files distribution

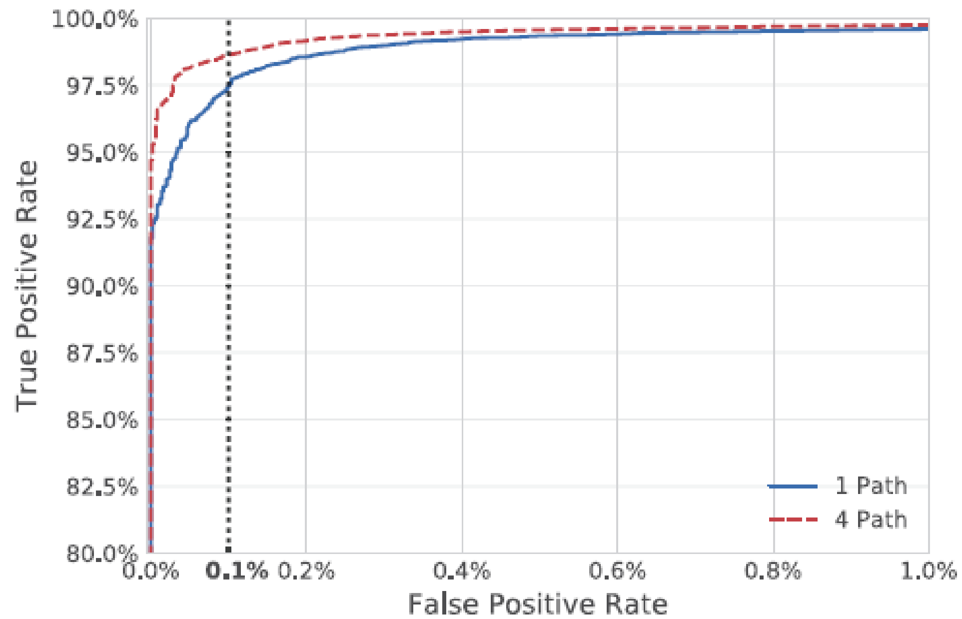
	Benign	Malware	Total	(%)
Training set	113,162	116,807	229,969	(70%)
Test set	49,254	49,310	98,564	(30%)

Classification

- Two layer funnel
  - Layer 1 – single path, 1 minute timeout
  - Layer 2 – 4 paths, timeout 1 minute

- Set FPR to 0.1%

ROC curve



FPR 0.1%
TPR 99.11%

Table 3: Classification funnel sample count

Layer#	Execution paths	Samples	(%)
1	1	94,845	(96.2%)
2	4	3,719	(3.8%)
Total		98,564	(100%)

How does packing go?

- Packed a bunch of benign stuff and saw what TAMALES said

Table 4: PE packing experiment results

Dataset	Packer	Samples	Predicted malware	(%)
DS1	*Unpacked*	13,000	0	(0%)
DS2	UPX	12,154	120	(0.98%)
DS3	VMProtect	11,783	518	(4.39%)
DS4	Themida	9,592	582	(6.06%)

TAMALES on Malware families

- Used system to classify into families with Decision Trees

Table 5: Malware family classification results of most common families

Label #	Malware family	Samples in test set	Accuracy
1	allaple	9195	99.98%
2	dinwod	4575	99.83%
3	virut	4213	96.14%
4	browsefox	2380	99.58%
5	parite	2012	99.02%
6	ramnit	1823	93.11%
7	multiplug	1437	99.93%
8	upatre	1187	98.02%
9	mira	1138	99.91%
10	loadmoney	864	98.86%
11	*unknown*	742	72.11%
12	linkular	714	100.00%
13	linkury	659	99.70%
14	elex	646	98.48%
15	onlinegames	511	85.17%
16	wajam	502	99.80%