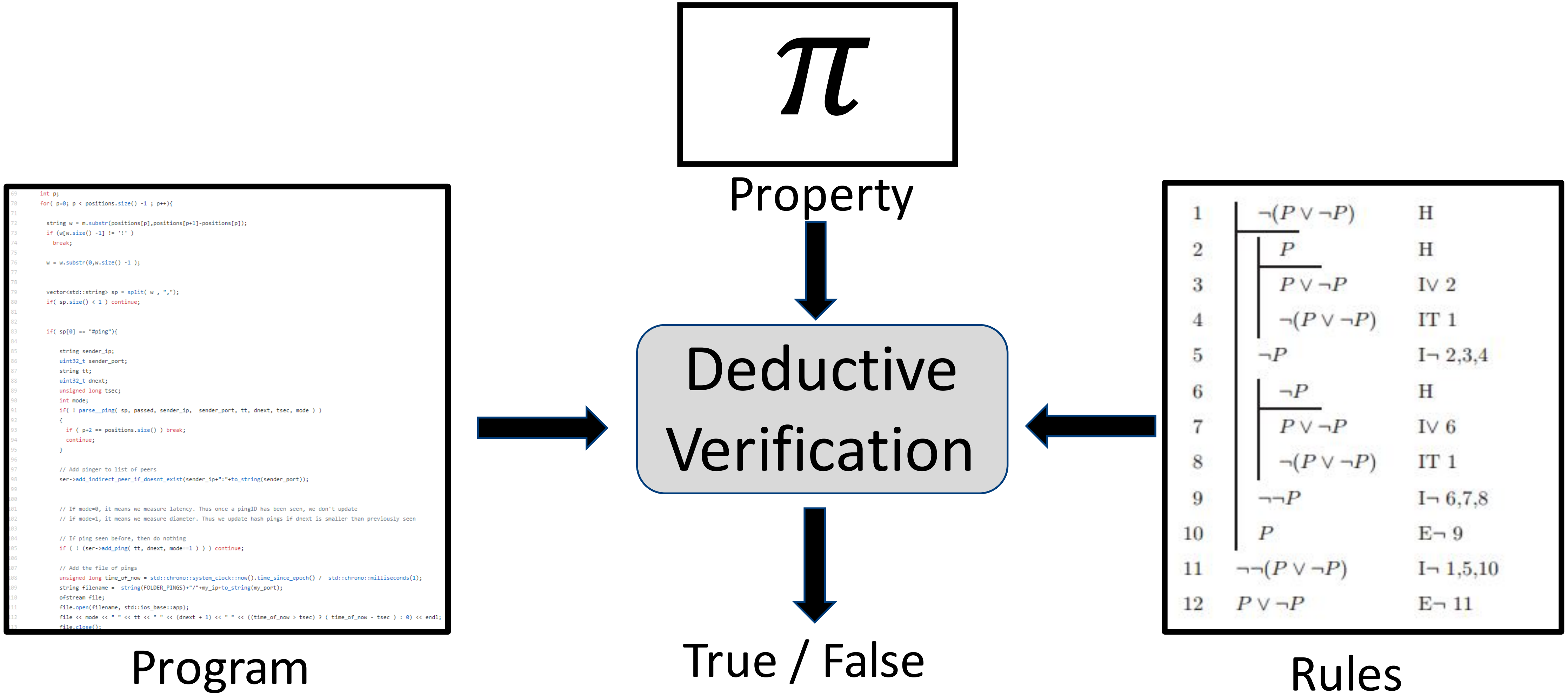
The Symbiosis of Program Analysis and Machine Learning

Prateek Saxena

Associate Professor

National University of Singapore

Program Analysis, Classically

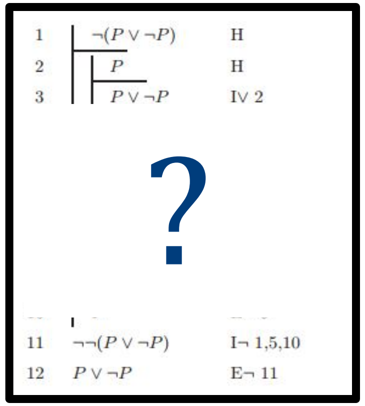


But, In Practice…

Program

?

Property



Rules

- Too Complex to Analyze / Model
- Probabilistic System

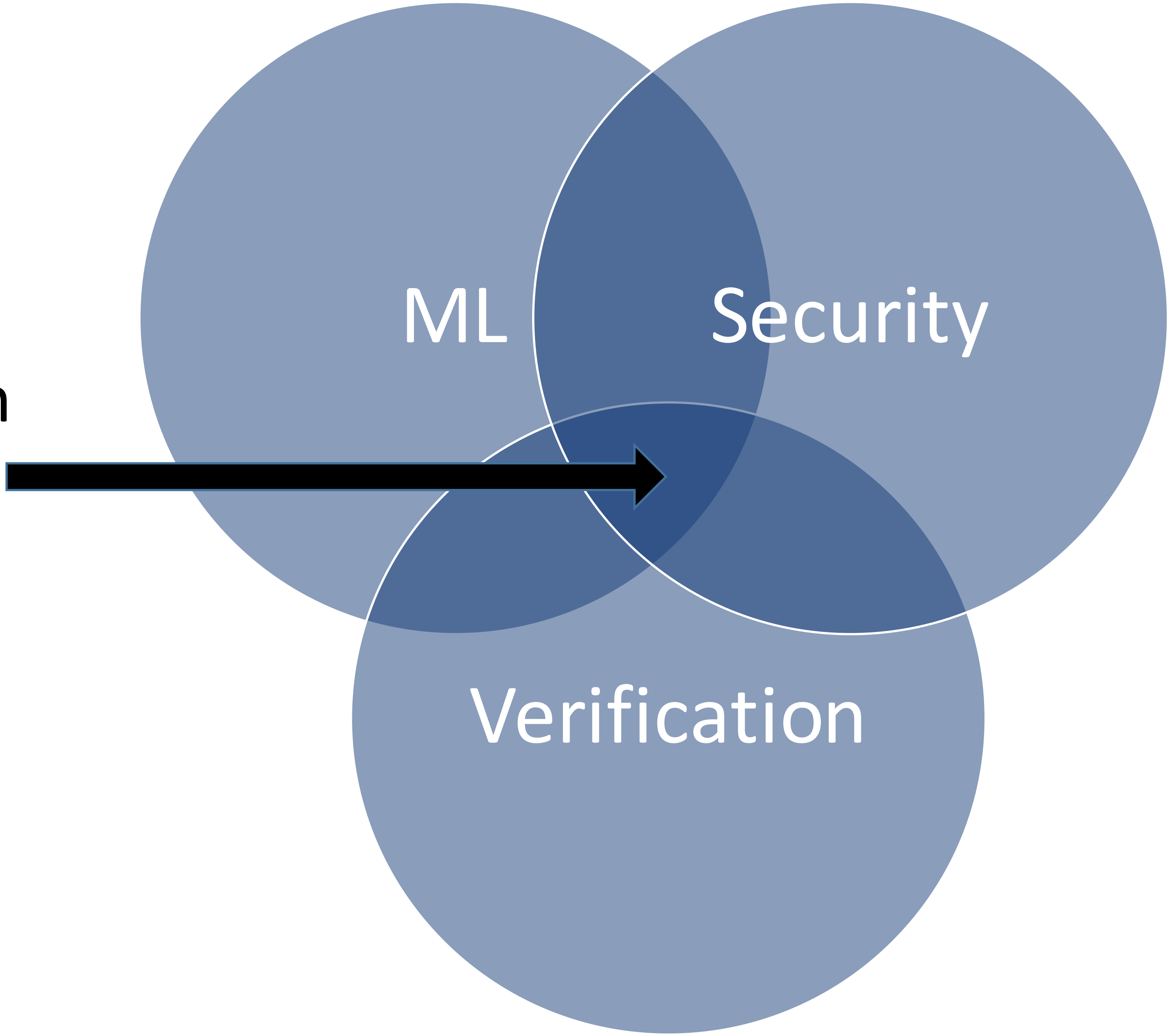
- Probabilistic / Stochastic Properties
- Ambiguous Spec. (Eg. Good patch?)

- Not Re-Targetable
- Intractable Analysis

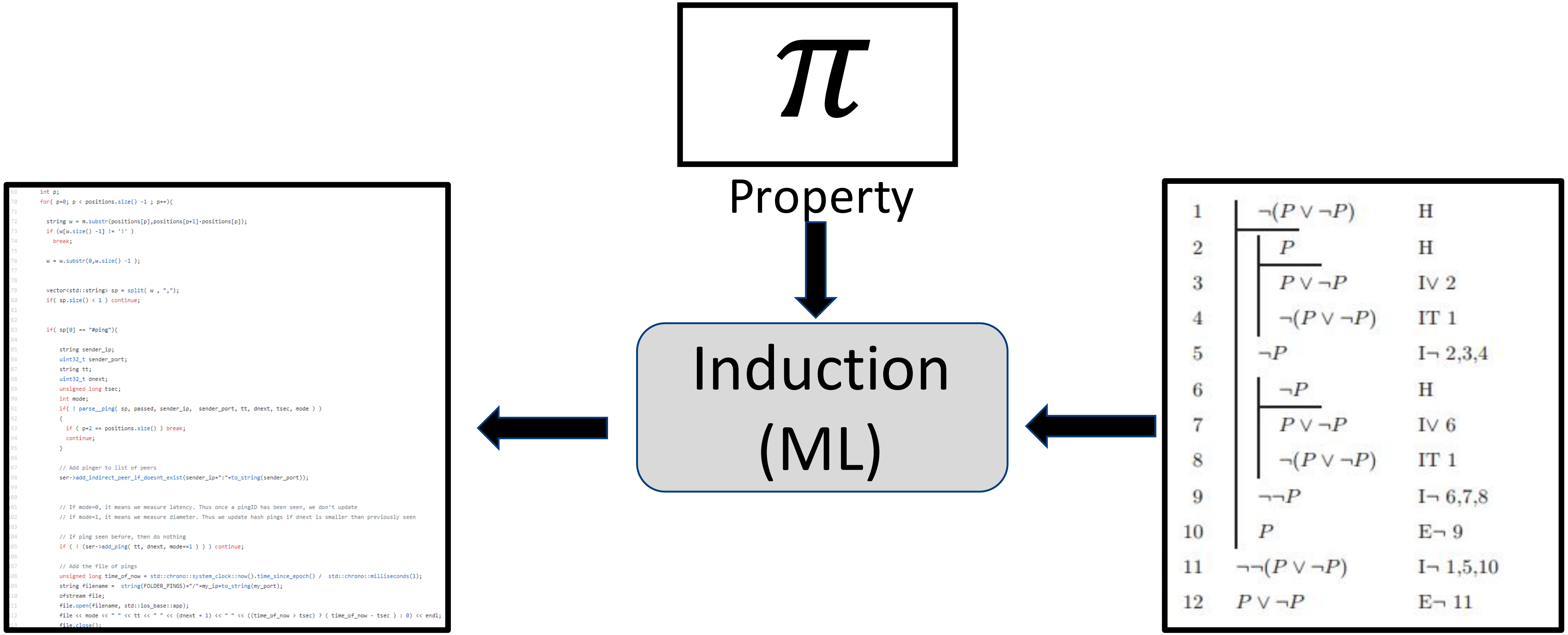
Can Machine Learning Help?

This Talk…

3 Mainstream Security Analysis Tasks



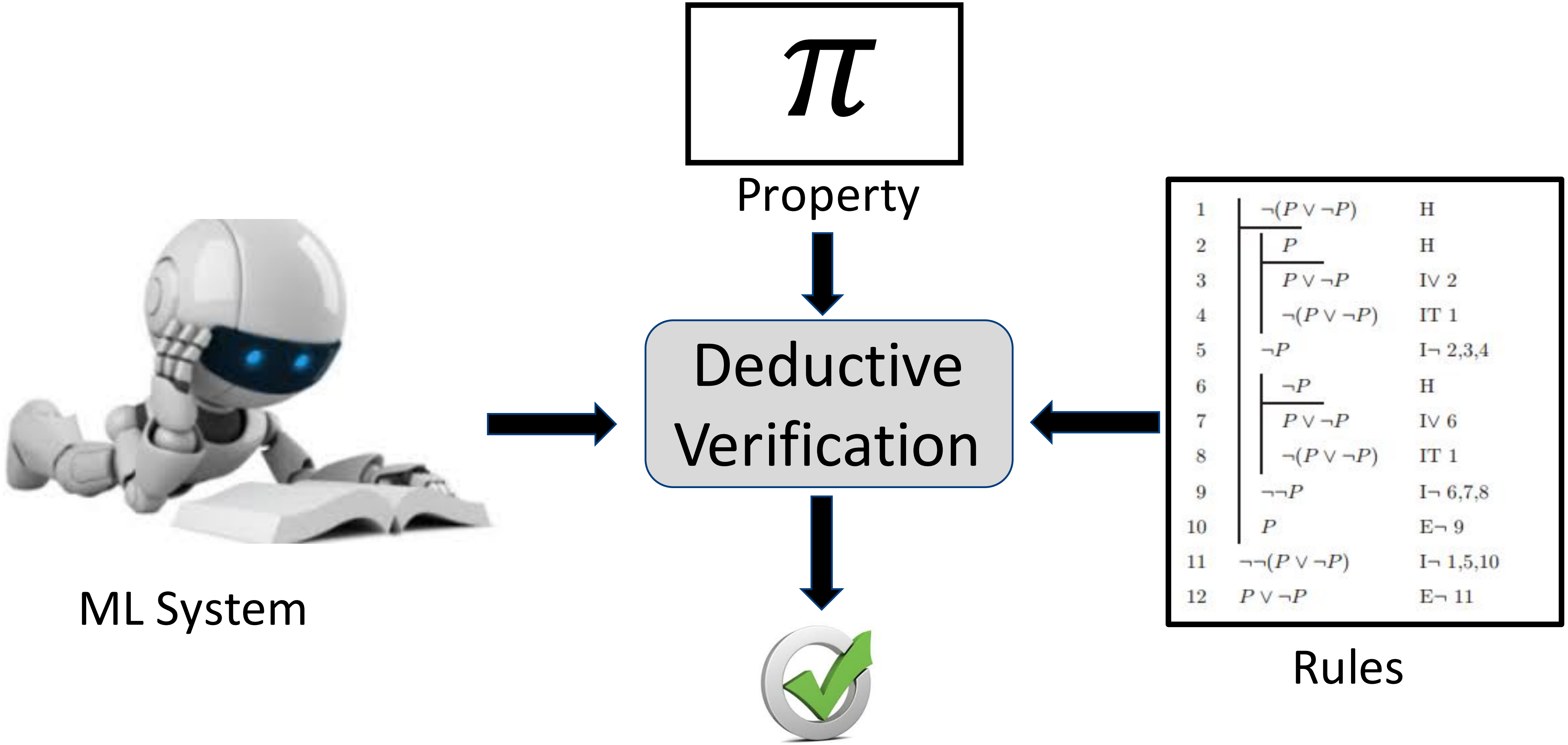
Machine Learning → Program Analysis
New Representations & Inference Tools



Program Representations

Rules

Program Analysis ➡ Machine Learning
Deductive Reasoning



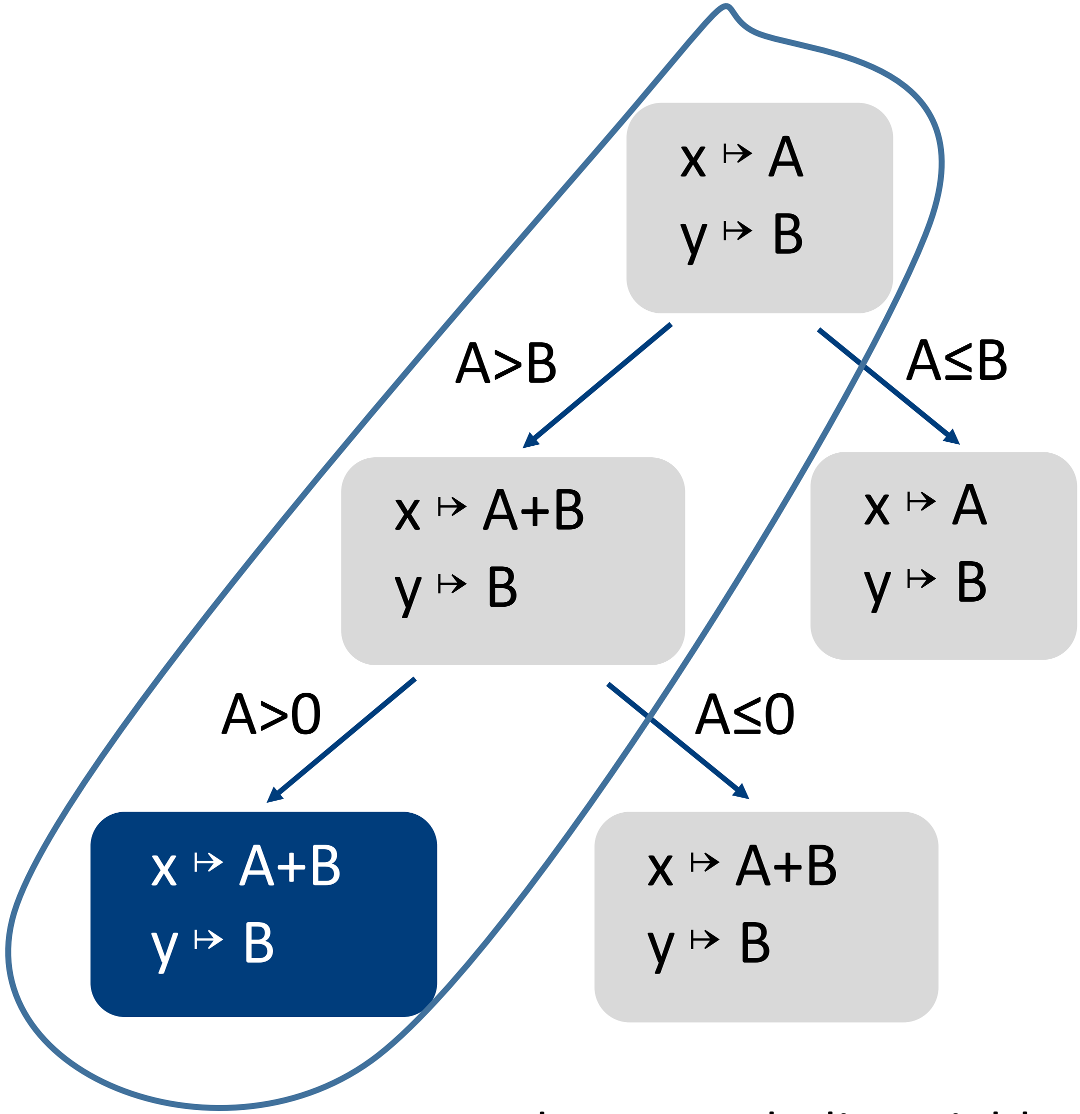
ML → PA:
New Representations & Inference Tools
For Symbolic Execution

Joint work with:
Shiqi Shen , Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury (NDSS 2019)

Symbolic Execution

```
def f (x, y):
 if (x>y)
   x = x+y
   if (x-y > 0)
     assert false
 return (x, y)
```

Dynamic Symbolic Execution (DSE):
A widely used variation of SE



x and y are symbolic variables
A and B are symbolic values

Symbolic Execution for Finding Security Bugs

KLEE

Pex

Angr

Kite

SAGE

jCUTE

Manticore

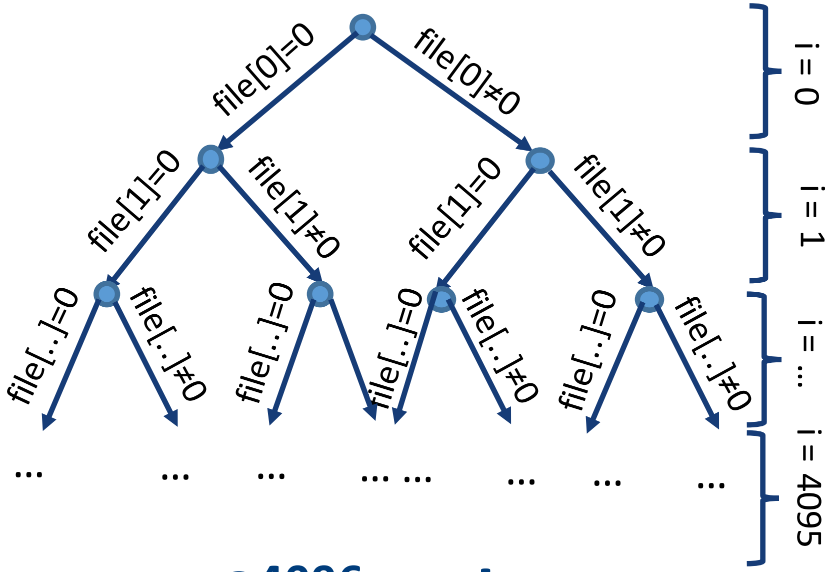
TRITON

Dynamic Binary Analysis

S²E

The Path Explosion Problem

```
void copy_data(..., int *file,...) {
    static double data[4096], value;
    read_double_value(file, ... );
    value = fabs (data [0]);
    for(i=0; i<4096; i++)
          if(file[i] == 0.0) count++;
    data[1] /= (value+count-3);
    …
}
```



2^{4096} paths

Prior Approaches: Learn a better representation, symbolically solve!

- Express in SMT theory of floating-point
- Infer that ‘count’ = # of 0s in input bytes
- Assert: value + count - 3=0

FPA | Vectors

Linear arithmetic

But, why choose this specific constraint representation?

```
void copy_data(..., int *file,...) {
    static double data[4096], value;
    read_double_value(file, ... );
    value = fabs (data [0]);
    for(i=0; i<4096; i++)
          if(file[i] == 0.0) count++;
    data[1] /= (value+count-3);
    …
}
```

FPA | Vectors

Linear arithmetic

BV

String

Real

Bool

UF

A Universal Approximate Representation?

Key Insights

Desired Representation:

$$count == \Sigma_{i\in[0,4095]}\ sign(file[i] == 0)$$

```
void copy_data(..., int *file..) {
    static double data[4096], value;
    read_double_value(file, ... );
    value = fabs (data [0]);
    for(i=0; i<4096; i++)
        if(file[i] == 0.0) count++;
    data[1] /= (value+count-3);
    …
}
```

A neural network is an approximate representation of the desired…

Remarks:

- Neural Networks are universal approximators
- Increasing practical success

Key Insights

Values of Symbolic Variables

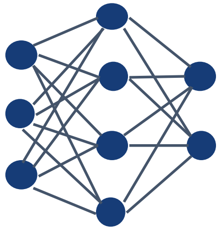
Values of Variables in CVP

Learn an approximation with small number of I/O examples

```
int main (…) {
  if (strlen(filename)>1 && filename[0]=='-')
    exit(1)
  copy_data(…);
  …
}
void copy_data(..., int *file ...) {
```

Approximate Constraint (as a neural net):

file



count & value

```
  data[1] /= (value+count-3);
  …
}
```

CVP: Divide-by-zero

A New Approach: Neuro-symbolic Execution

Program

π

Property

Symbolic (SMT) Constraints

Neural Network

Symbolic Exploit Condition

SATISFIABLE?

Neuro-Symbolic Execution – Shen et al. [NDSS'19]

Constraint Solving: Satisfiability Checking

1. Reachability constraints:

$$strlen(filename) \leq 1$$
$$\vee \ filename \neq '-'$$

Purely symbolic constraints: → SMT solver

No variable shared with neural constraints

$$\wedge$$

$$N\!: infile \rightarrow (value, count)$$

2. Vulnerability condition:

$$value + count - 3 == 0$$

Mixed constraints:

Including both neural constraints and symbolic constraints with shared variables

Neuro-Symbolic Execution – Shen et al. [NDSS'19]

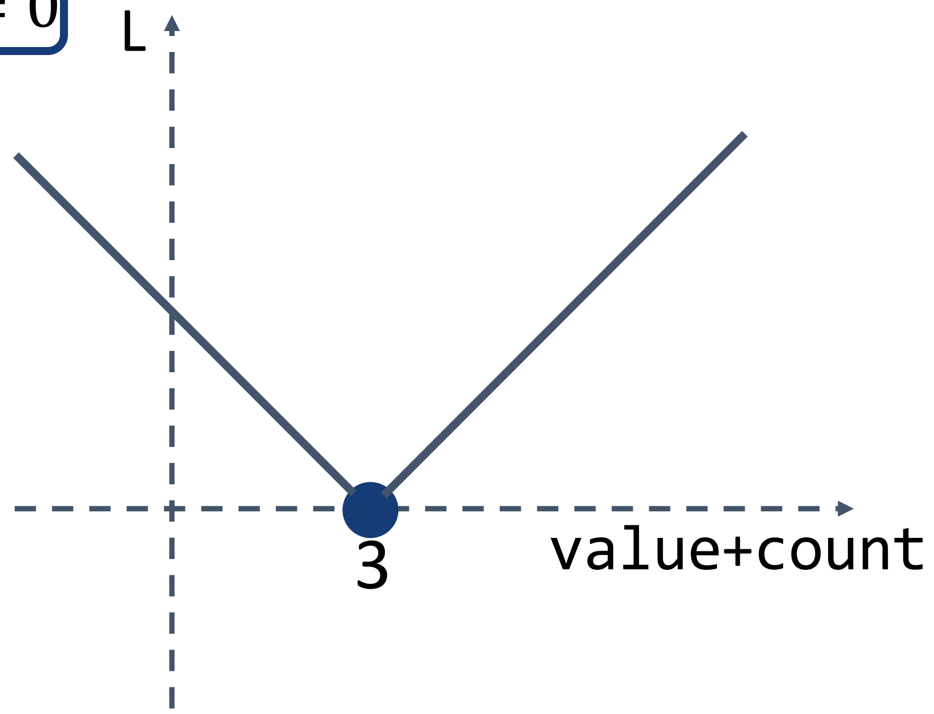
Solving SMT + Neural Constraints: Encode SMT constraints as the loss function

$N: infile \rightarrow (value, count) \wedge \boxed{value + count - 3 == 0}$

Symbolic constraint

Criterion for crafting the loss function:
The minimum point of the loss function satisfies the symbolic constraints.

$$L = abs(value + count - 3)$$

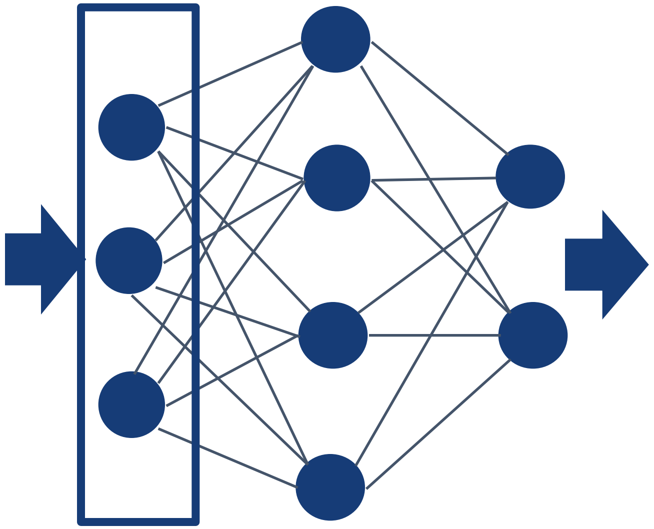


Neuro-Symbolic Execution – Shen et al. [NDSS'19]

Optimize using Gradient Descent

Gradient: $\nabla_{file} L$

file



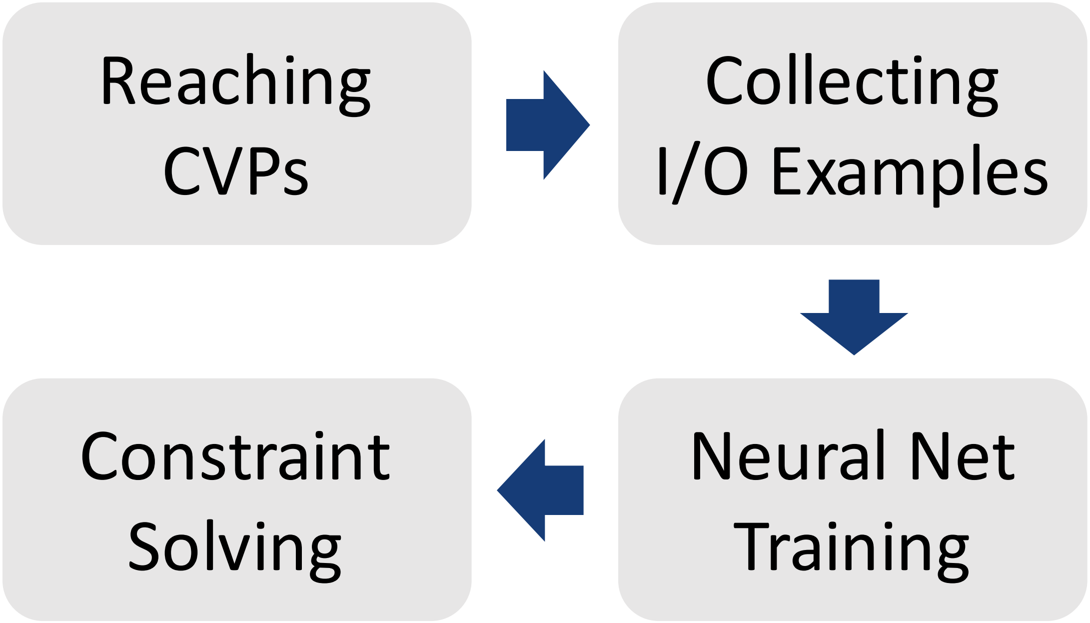
count,value

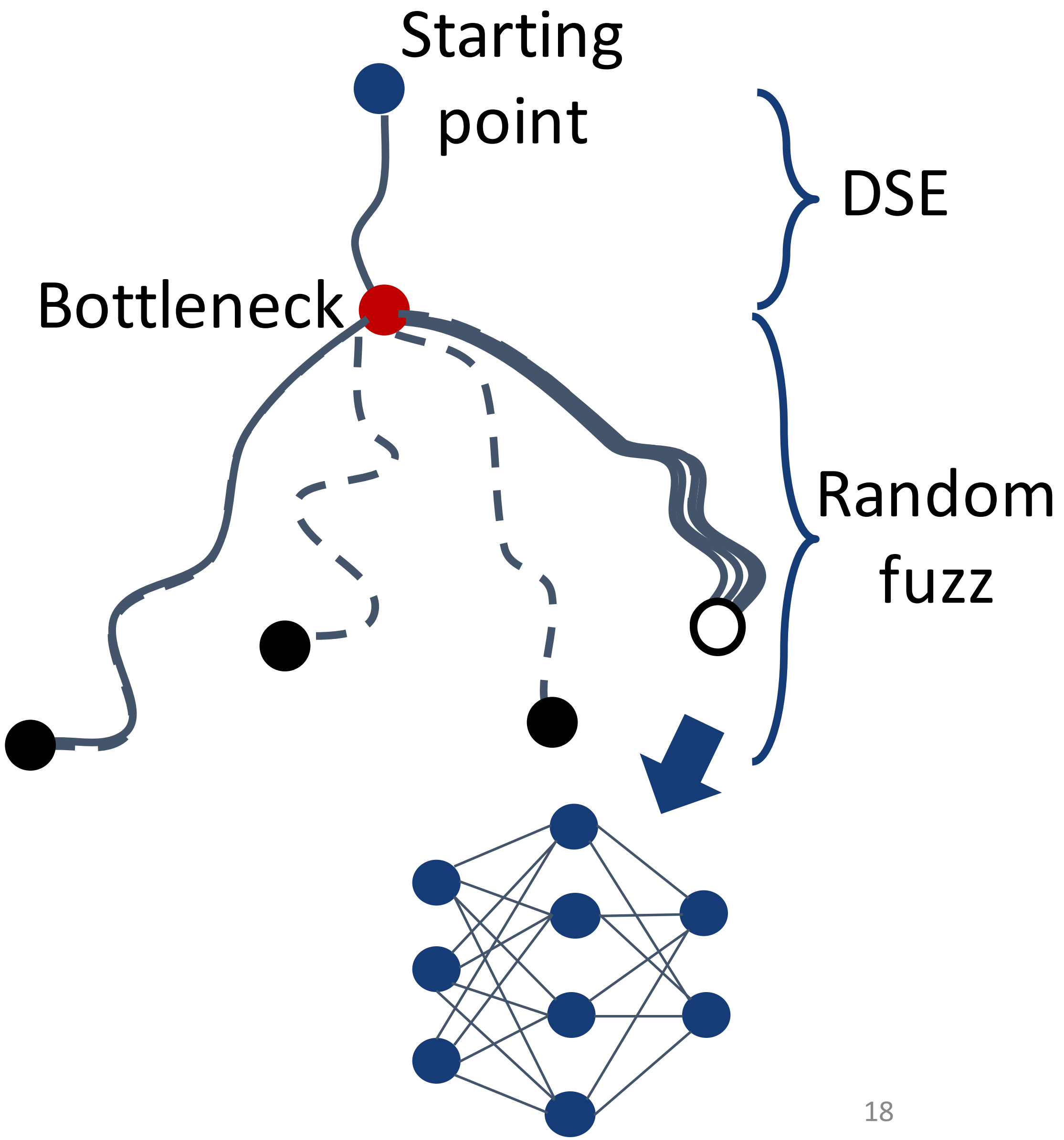
count	value	loss
257	20	274
1	20	18
0	20	17
…	…	…
0	3	0

Concretely validate the exploit

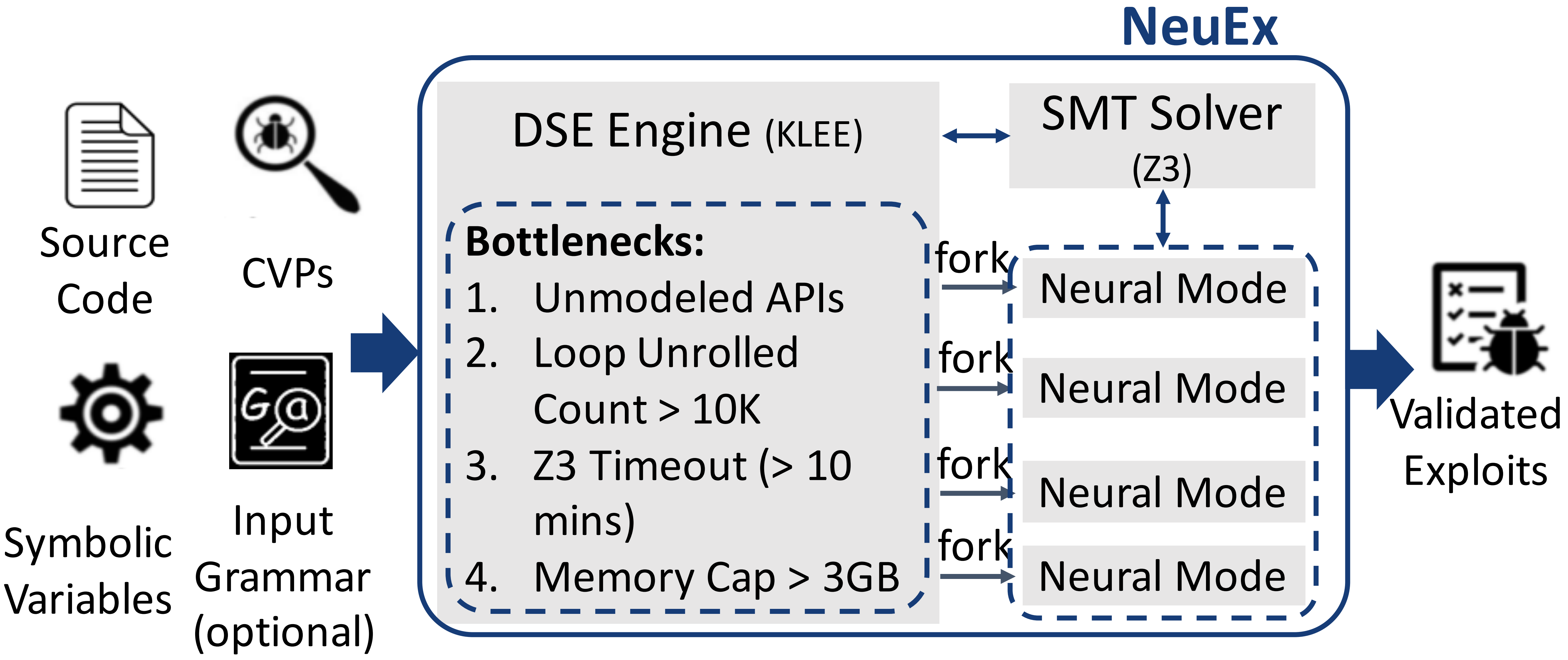
file: 000…

NeuEX = Neuro-symbolic Execution + KLEE





NeuEx Tool Overview



Neuro-Symbolic Execution – Shen et al. [NDSS'19]

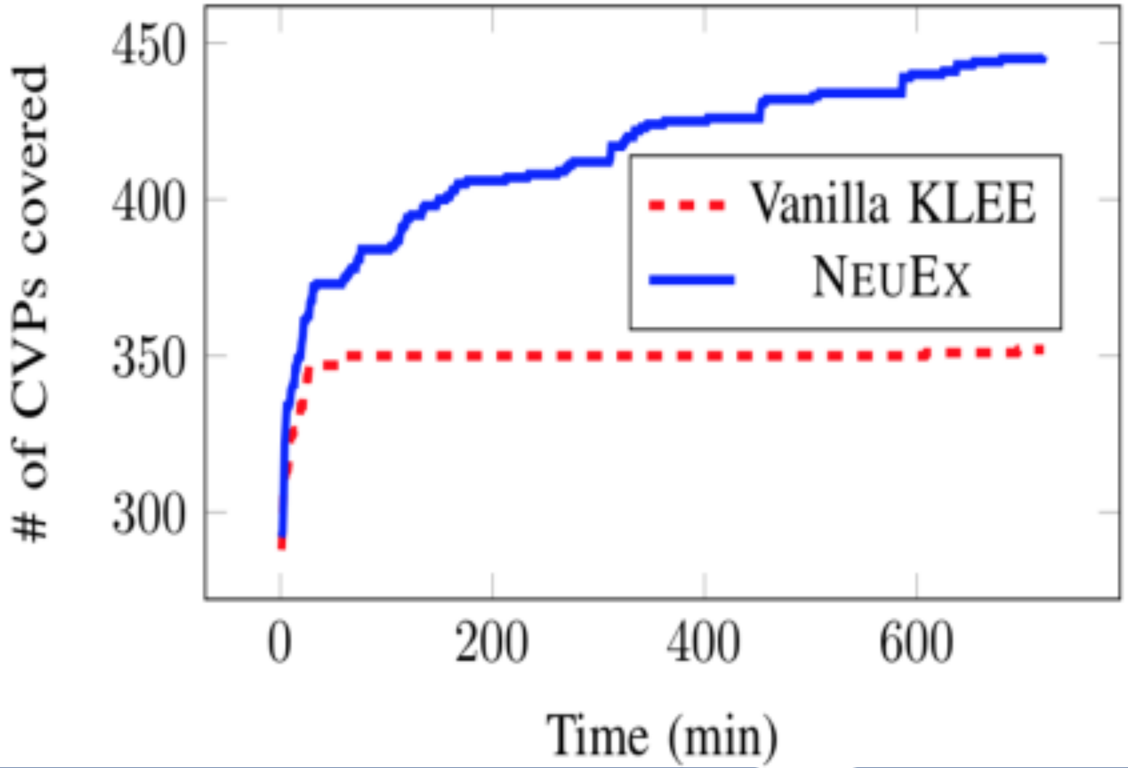
Evaluation

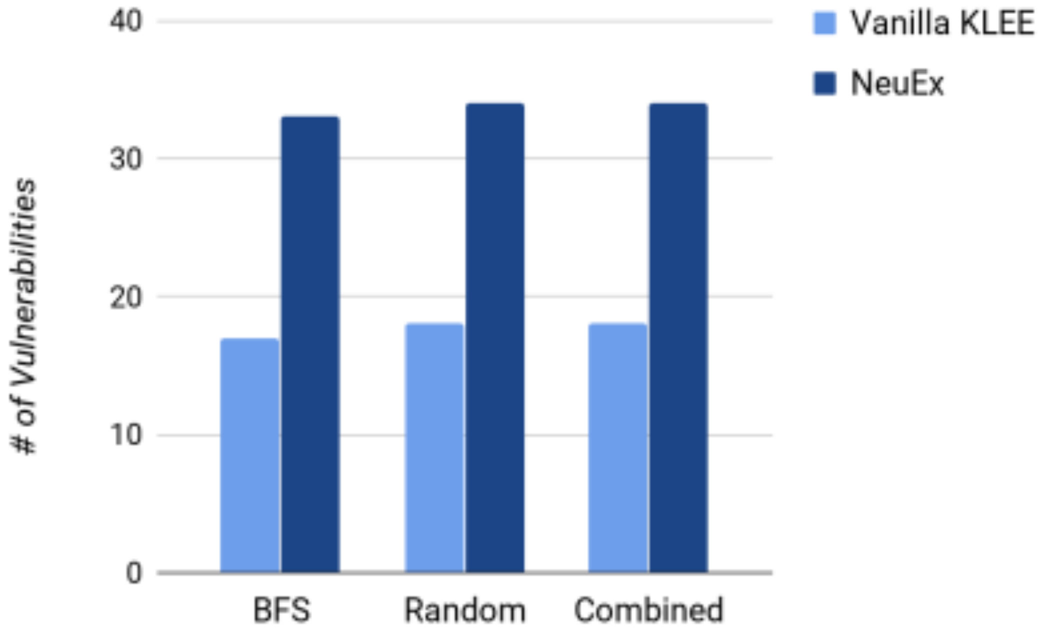
- **Recall**: Neural mode is only triggered when DSE encounters bottlenecks
- **Benchmarks**: 7 Programs known to be difficult for classic DSE
  - 4 Real programs
    - cURL: Data transferring
    - SQLite: Database
    - libTIFF: Image processing
    - libsndfile: Audio processing
  - LESE benchmarks
    - BIND, Sendmail, and WuFTP

Include:
1. Complex loops
2. Floating-point variables
3. Unmodeled APIs

Neuro-Symbolic Execution – Shen et al. [NDSS'19]

NeuEx vs KLEE





#CVPs reached or covered by NeuEx is 25% higher than vanilla KLEE.

KLEE gets stuck (e.g. complex loops)

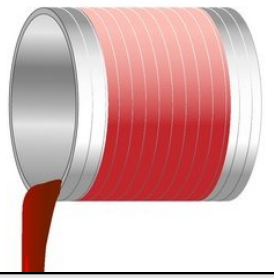
NeuEx finds 94% and 89% more bugs than vanilla KLEE in BFS and RAND mode in 12 hours.

Neuro-Symbolic Execution – Shen et al. [NDSS'19]

ML $\longrightarrow$ PA:
New Representations & Inference
For Taint Analysis

Joint work with:
Shiqi Shen , Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury (NDSS 2019)

Taint Analysis



- Taint analysis tracks the information flow within a program:
  - E.g. T[v] is the taint bit for operand "v"
- Taint analysis is the basis for many security applications
  - Information leakage detection
  - Enforcing program integrity
  - Vulnerability detection
  - …

```
int parse_buffer(char buffer[100], struct
pkt_info *info) {
    char check_flag;

    check_flag = buffer[5] & 0x16;

    err = init_pkt_info(info);
    if (!err)
        return err;
    info->flag = check_flag;
    /* … */
    strncpy(info->data, buffer + 6);
    info->seq = get_current_seq();
    return OK;
}
```

Is Return Address Tainted?

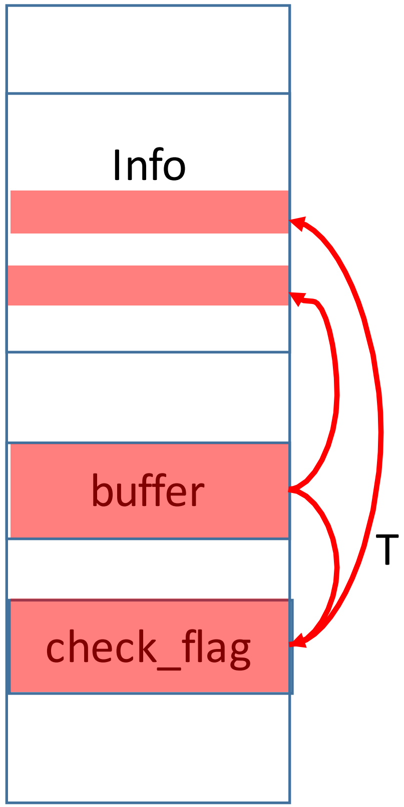
Taint Analysis on Binaries

```
/* tainted input from network socket */
int parse_buffer(char buffer[100], struct
pkt_info *info) {
    char check_flag;

    check_flag = buffer[5] & 0x16;

    err = init_pkt_info(info);
    if (!err)
        return err;
    info->flag = check_flag;
    /* … */
    strncpy(info->data, buffer + 6, 50);
    info->seq = get_current_seq();
    return OK;
}
```

Taint Map
T[ ]



T[check_flag] = T[buffer+5]

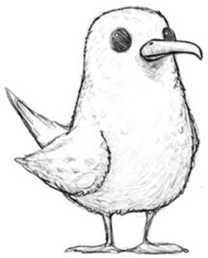
Write binary taint rules based on instruction semantics

```
movsx   eax, byte ptr [rsi + 5]
and     eax, 16
mov     cl, al
mov     byte ptr [rbp - 25], cl
```

Taint Rule Representations in Existing Systems

- What is the taint rule for `and eax, 16` on the x86 architecture?

Taint Engine 1	Taint Engine 2	Taint Engine 3
T[eax] = T[eax]	T[eax] = T[eax]	T[eax] = T[eax]
	T[pf] = T[sf] = T[zf] = T[eax] T[of] = T[cf] = 0	T[pf] = T[sf] = T[zf] = T[eax] T[of] = T[cf] = 0
		if imm == 0 { T[eax] = 0 }



One Engine To Serve'Em All: Leong et Al. [NDSS'19]

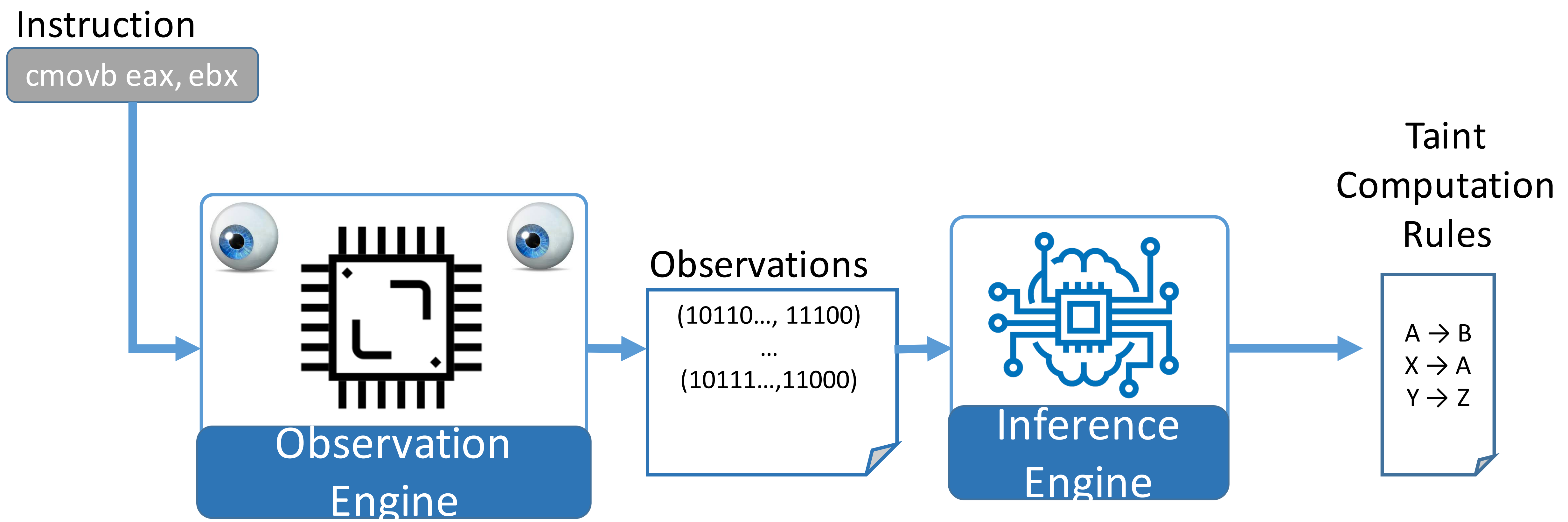
Complexity of “Real” Taint Rules

- Input dependent propagation
- Size dependent propagation
- Architectural quirks for backwards compatibility
- There can be 1000s of opcodes per instruction set!

```
if (size == 64 || size == 32 || size == 16) {
  for (x = 0; x < size / 8; x++) {
    if (t1[x] & t2[x]) t1[x] = 1;
    else if (t1[x] and !t2[x])
      t1[x] = t1[x] & op2[x];
    else if (!t1[x] & t2[x])
      t1[x] = t2[x] & op1[x];
    else t1[x] = 0;
  } else if (size == 8) {
// 0 if it's lower 8 bits, 1 if it's upper 8 bits
    pos1 = isUpper(op1); pos2 = isUpper(op2);
    if (t1[pos1] & t2[pos2]) t1[pos1] = 1;
    else if (t1[pos1] & !t2[pos2])
      t1[pos1] = t1[pos1] & op2[pos2];
    else if (!t1[pos1] & t2[pos2])
      t1[pos1] = t2[pos2] & op1[pos1];
    else t1[pos1] = 0;}}
if (mode64bit == 1 and size == 64)
  for (x = 32; x < size; x++) t1[x] = 0;
```

One Engine To Serve’Em All: Leong et Al. [NDSS’19]

Learning Taint Rules Automatically

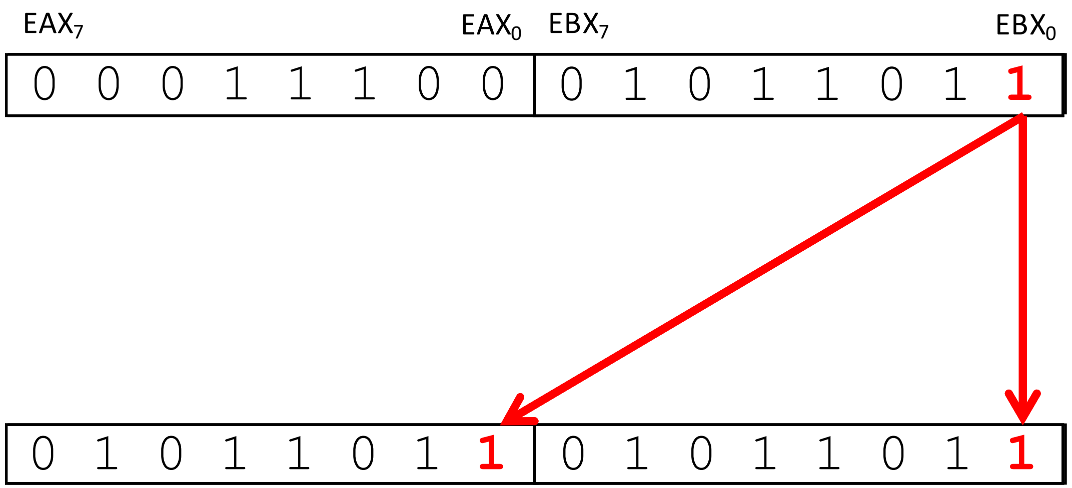


One Engine To Serve'Em All: Leong et Al. [NDSS'19]

TaintInduce: Sample and Learn

- Flip a bit and observe the output for changes.
  - $\Delta EBX_0 \rightarrow \Delta\, EAX_0$
  - $\Delta EBX_0 \rightarrow \Delta\, EBX_0$
- Influence (Inf) only valid if :
  - EAX = 11100011, EBX = 00101000
- Form a truth table with all of the collected observations.
  - True if there is a change, False otherwise
- Unseen values are conservatively set to “Don’t-Cares”

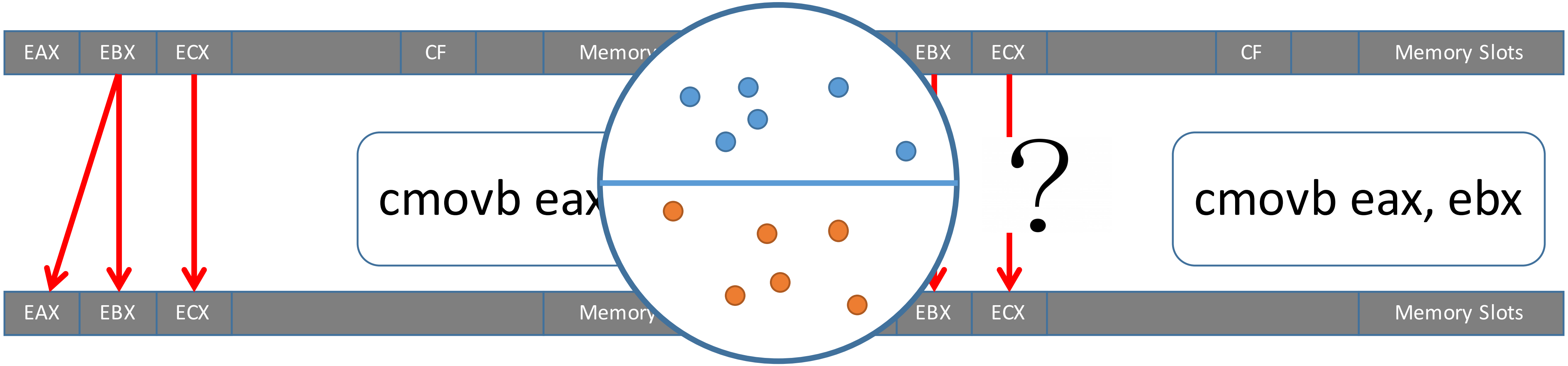
mov eax, ebx



EAX_0	EAX_1	…	EBX_0	EBX_1	…	Inf
1	1	…	0	0	…	1
1	1	…	1	0	…	1
0	0	…	1	1	…	1
0	0	…	0	0	…	1
…	…	…	…	…	…	0

One Engine To Serve’Em All: Leong et Al. [NDSS’19]

Captures Conditional & Indirect Dependencies



ebx → eax	eax → eax
CF=1, EAX=542, EBX=19, ECX=7, …	CF=0, EAX=12, EBX=4, ECX=1023…
CF=1, EAX=32, EBX=3, ECX=0, …	CF=0, EAX=42, EBX=11, ECX=13, …
CF=1, EAX=873, EBX=32, ECX=1, …	CF=0, EAX=2, EBX=3, ECX=33, …
…	…

One Engine To Serve'Em All: Leong et Al. [NDSS'19]

TaintInduce: Outputs a succinct rule

- Use inference technique to learn a **succinct rule** for the observed function

CF=0, EAX=12, …Z	False
CF=1, EAX=333, …	True
CF=0, EAX=42, …	False
CF=0, EAX=44, …	False
CF=1, EAX=873, …	True
CF=0, EAX=1023, …	False
CF=0, EAX=33, …	False
CF=1, EAX=32, …	True
CF=0, EAX=2, …	False
…	DC

Inference →

IF

CF=1	True

THEN $(EBX_0 \rightarrow EAX_0)$

ELSE $(EAX_0 \rightarrow EAX_0)$

One Engine To Serve'Em All: Leong et Al. [NDSS'19]

Results: Comparison with State-of-the-art

Matches: 93.27% - 99.5% with existing hand-written tools.

Only 0.28% discrepancies are errors in TaintInduce.

X86 Instructionsxw	Arith	Comp	Jump	Move	Cond	FPU	SIMD	Misc	Total
TaintInduce	43	9	33	33	60	85	259	28	550
libdft	15	5	1	30	32	X	X	8	91
Triton	38	9	19	33	32	X	144	13	288
TEMU	7	1	2	3	X	X	X	X	13

One Engine To Serve'Em All: Leong et Al. [NDSS'19]

Results: Coverage and Correctness

Auto-generated taint rules for 4 architectures: x86, x64, AArch 64, MIPS-I

with no mistakes for ~71% of the instructions

Methodology: train for 100 seeds, test on 1000 random inputs for each instruction

	Arith	Comp	Jump	Move	Cond	FPU	SIMD	Misc
x86	√	√	√	√	√	√	√	√
x64	√	√	√	√	√	√	√	√
AArch64	√	√	√	√	√	√	√	√
MIPS-I	√	√	√	√	-	-	-	-

Room for Future Work: Learn precise rules for all the instructions….

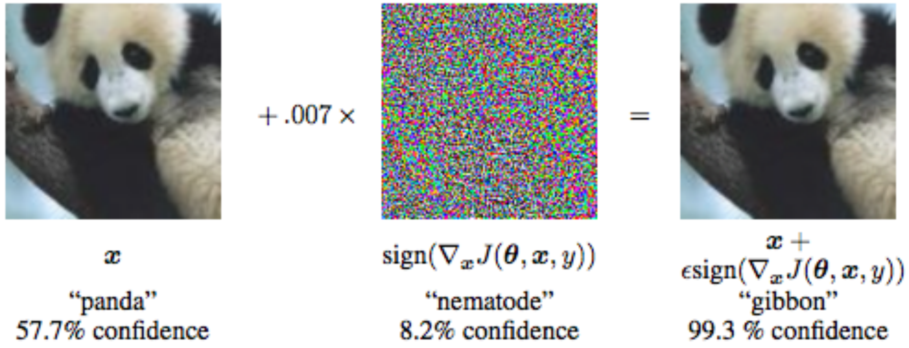
One Engine To Serve'Em All: Leong et Al. [NDSS'19]

PA $\Rightarrow$ ML: Deductive Reasoning

Joint work with:
Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel (CCS 2019)_

Concerns with ML Systems

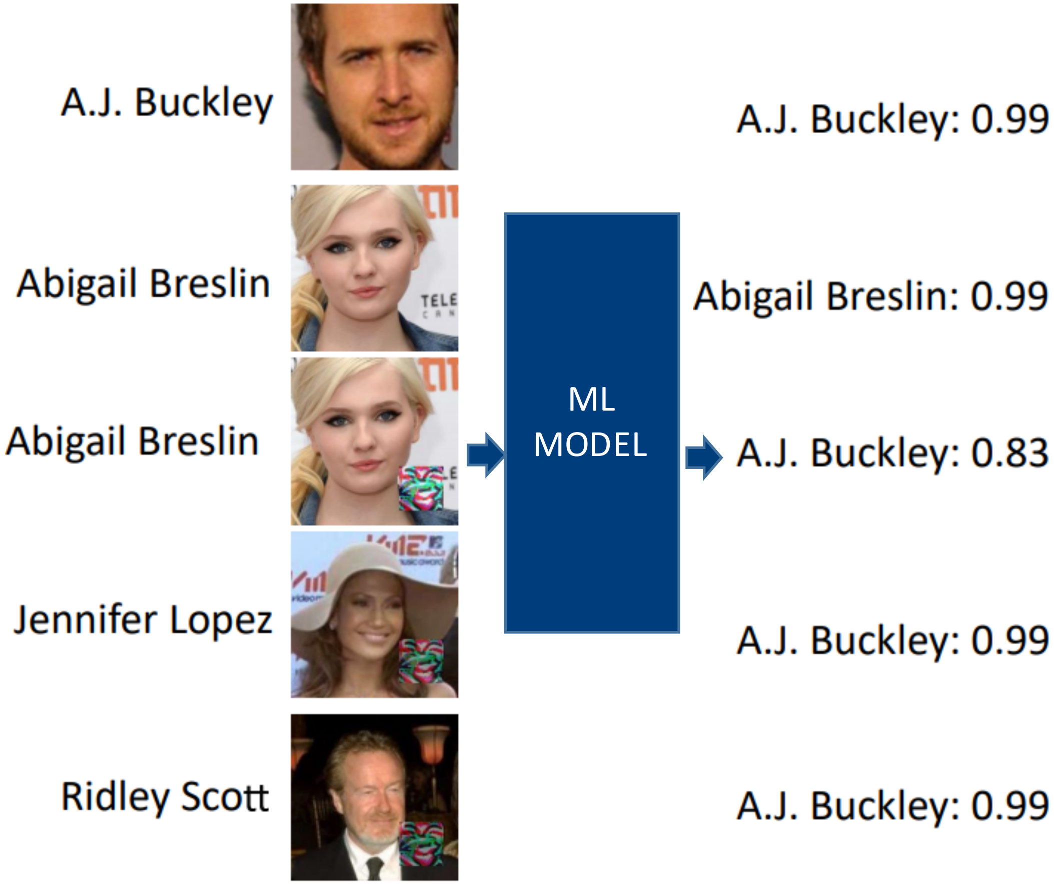
Robustness



[Adversarial Examples – Goodfellow et al.]

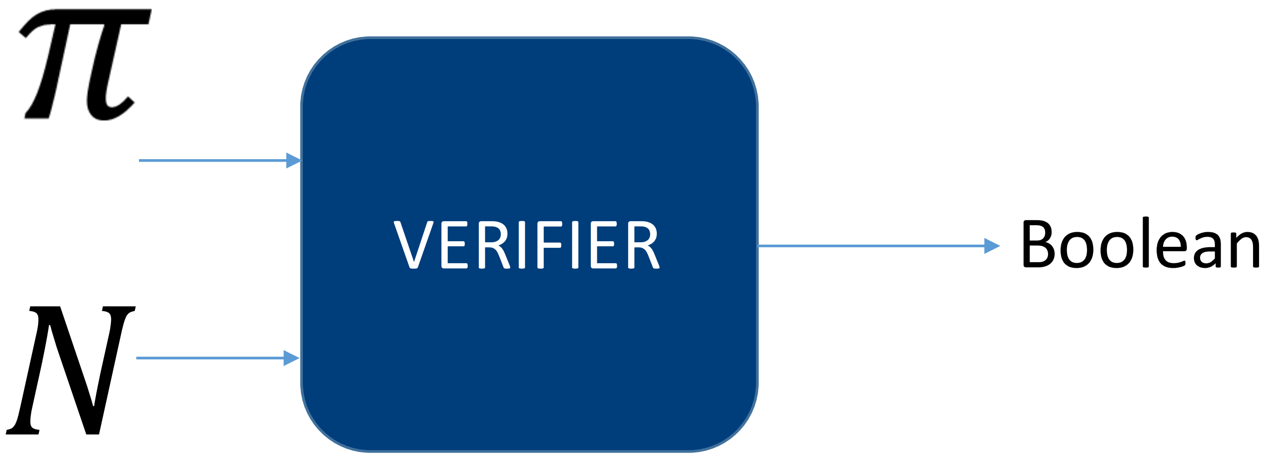
Fairness

Memorization



[Trojaning Attacks - Liu et. al]

Qualitative Verification



$\exists x, \pi(N, x)$

But, the network or property is often stochastic…

Quantitative Verification

Quantitative Verification of Neural Networks And its Security Applications – Baluta et al. [CCS’19]

NPAQ: A Quantitative Verifier For Neural Nets

π → COUNT-PRESERVING ENCODERS

N → COUNT-PRESERVING ENCODERS

COUNT-PRESERVING ENCODERS $\xrightarrow[(\varphi)]{\text{CNF FORMULA}}$ APPROX. MODEL-COUNTER → COUNT $|R(\varphi)|$

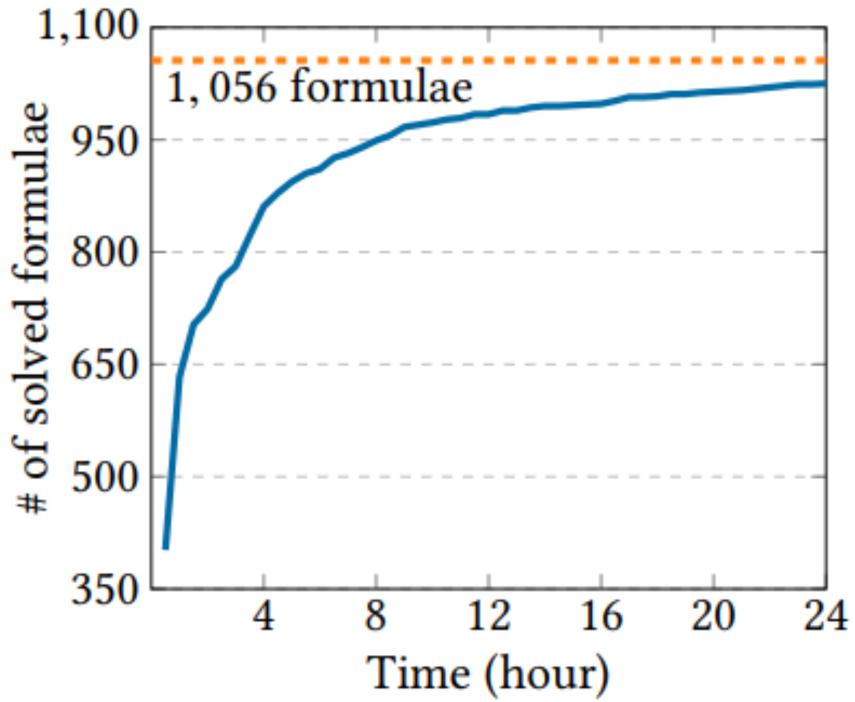
PAC-style Soundness Guarantees:

$$\boxed{\Pr[(1+\underbrace{\epsilon}_{\text{Error Tolerance}})^{-1}\underbrace{|R(\varphi)|}_{\text{True Count}} \leq \underbrace{r}_{\text{Approx. Count}} \leq (1+\epsilon)|R(\varphi)|] \geq 1-\underbrace{\delta}_{\text{Confidence}}}$$

Quantitative Verification of Neural Networks And its Security Applications – Baluta et al. [CCS'19]

NPAQ Results

- 84 models of up to 51,410 parameters
- 1,056 encoded CNF formulae
- 3 applications on MNIST and UCI Adult datasets:
  - **Robustness:** How many adversarial samples within some perturbation distance?
  - **Fairness:** How often will a prediction change (favorably) if the gender of the applicant is changed, keeping all else constant?
  - **Trojan attack efficiency:** How often will an image with a trigger result in desired misclassification?



97.1% of the encoded formula solved within 24 hours timeout each

Quantitative Verification of Neural Networks And its Security Applications – Baluta et al. [CCS'19]

Key Takeaways

- **Machine Learning Helps Program Analysis**
  - When program, property or analysis rules are uncertain
  - Provides powerful approximate representations and solving tools
  - Specific Applications:
    - Neuro-Symbolic Execution
    - Automatically Learning Taint Rules

- **Program Analysis Helps Machine Learning**
  - By verifying properties
  - SAT/SMT-based quantitative reasoning is a powerful tool
  - Specific Applications: Fairness, Robustness, Memorization

Thank you!