PELICAN: Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An,
Qiuling Xu, Yingqi Liu, Yapeng Ye, Yaoxuan Wu, Xiangyu Zhang

PURDUE UNIVERSITY

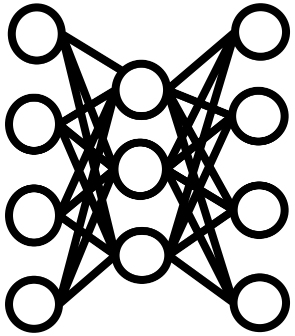
August 9, 2023

Deep Learning for Binary Analysis

Pelican

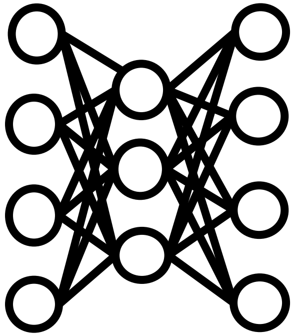
Deep Learning for Binary Analysis

```
1010101010
1101101010
    … …
0101010000
```



Deep Learning for Binary Analysis

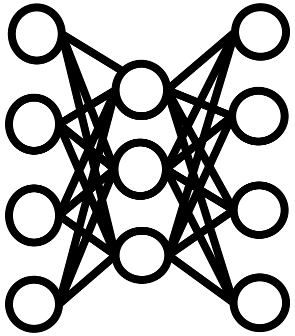
```
1010101010
1101101010
… …
0101010000
```



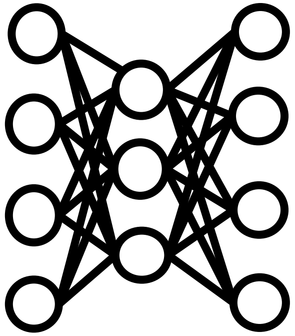
```
mov    rdi, [rdi + rax]
mov    rsi, [rdi]
mov    [rsi + 8], rdi
pop    esi
ret
```

Deep Learning for Binary Analysis

```
1010101010
1101101010
… …
0101010000
```



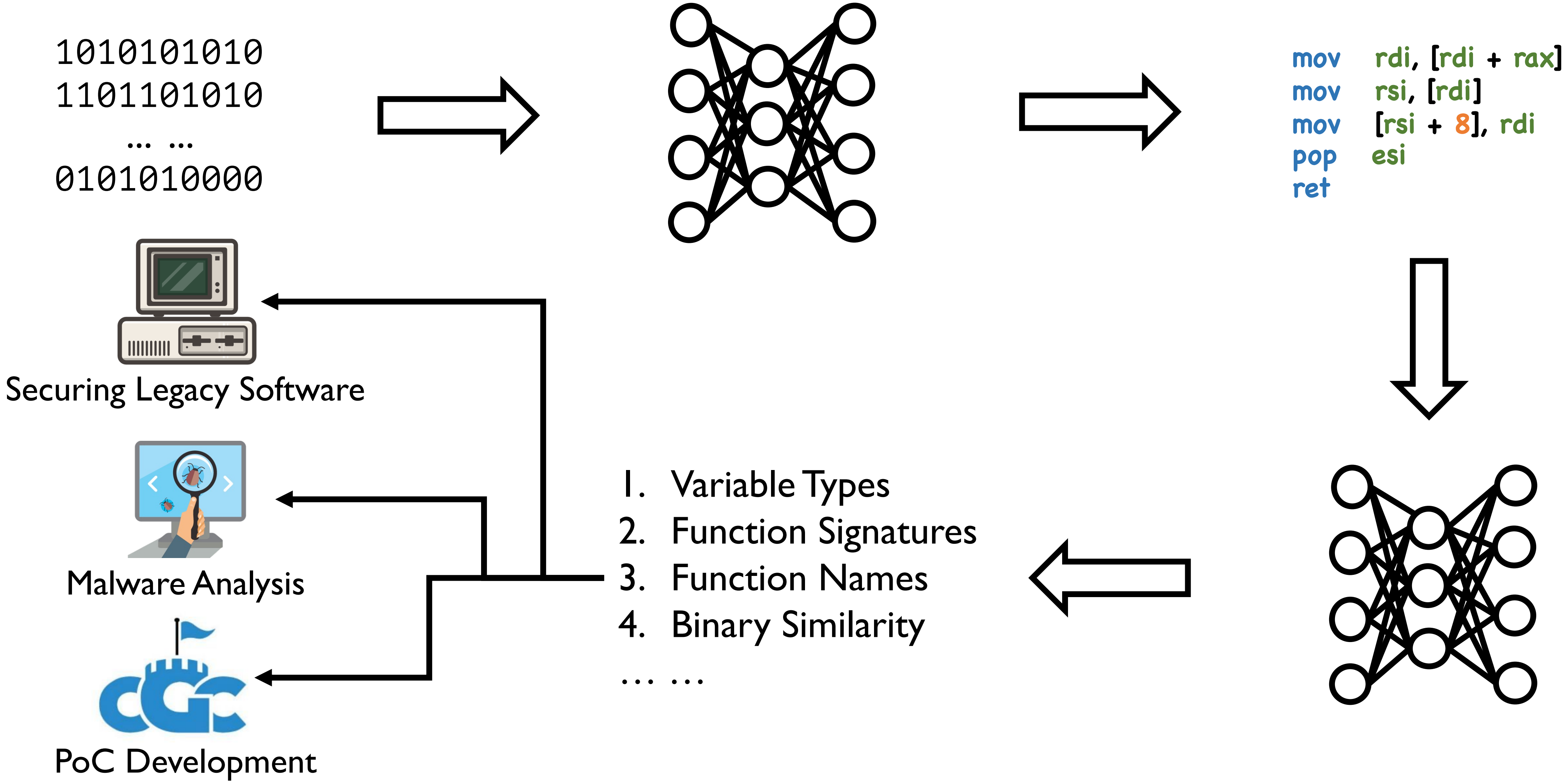
```
mov   rdi, [rdi + rax]
mov   rsi, [rdi]
mov   [rsi + 8], rdi
pop   esi
ret
```



1. Variable Types
2. Function Signatures
3. Function Names
4. Binary Similarity

… …

Deep Learning for Binary Analysis

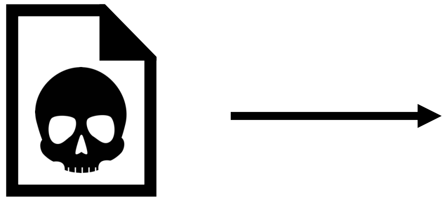


Key Question

Are these binary analysis models sufficiently robust against carefully manipulated input binaries?

Key Question

Are these binary analysis models sufficiently robust against carefully manipulated input binaries?



`EncryptAllFiles`

Ransomware

Key Question

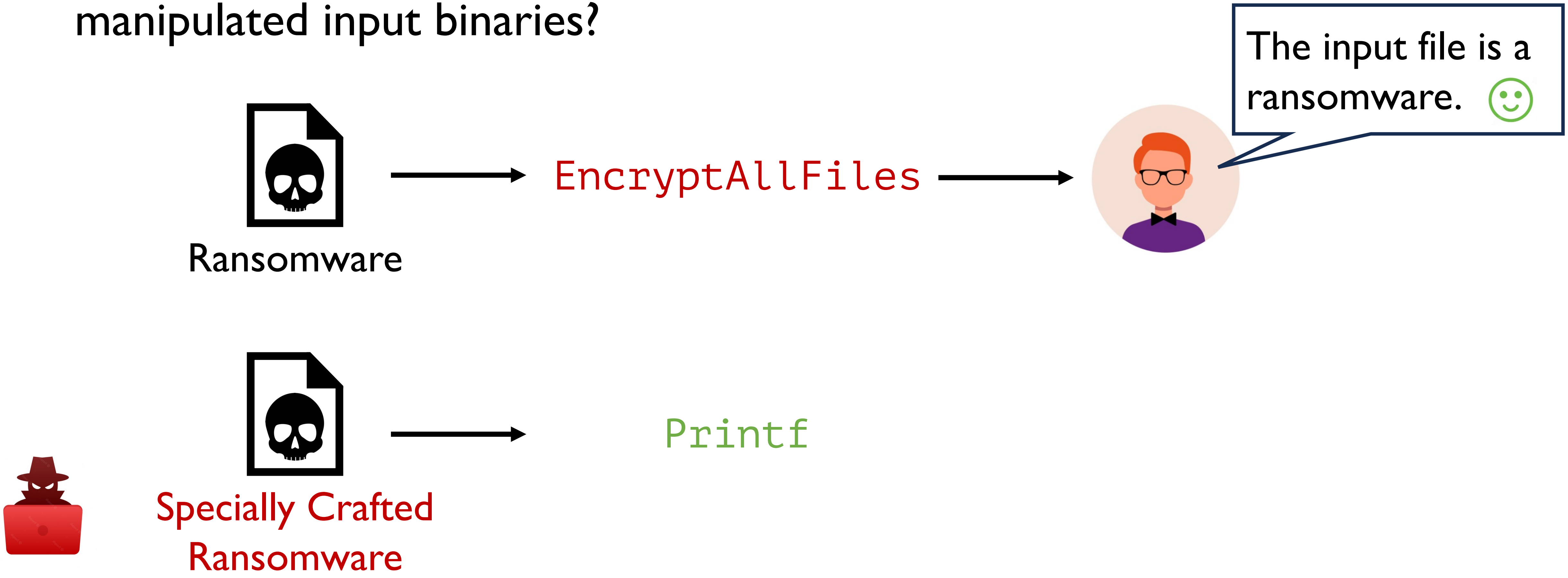
Are these binary analysis models sufficiently robust against carefully manipulated input binaries?

Key Question

Are these binary analysis models sufficiently robust against carefully manipulated input binaries?

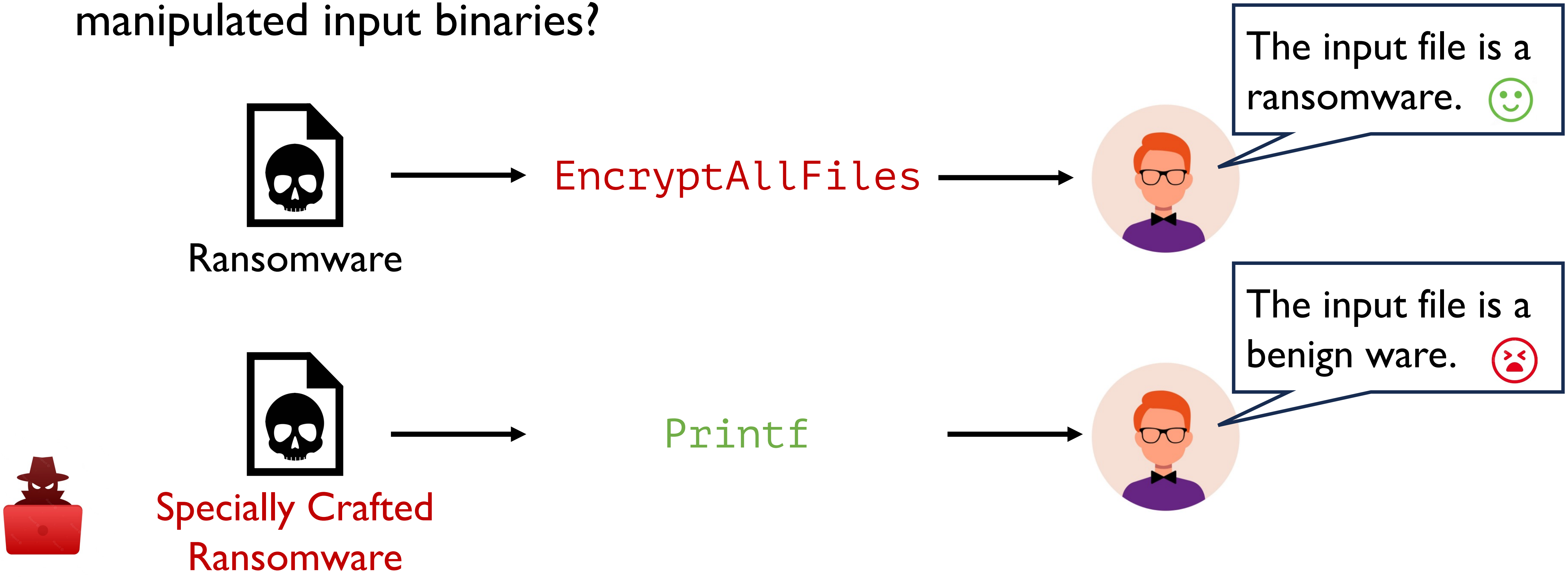
Key Question

Are these binary analysis models sufficiently robust against carefully manipulated input binaries?



Key Question

Are these binary analysis models sufficiently robust against carefully manipulated input binaries?

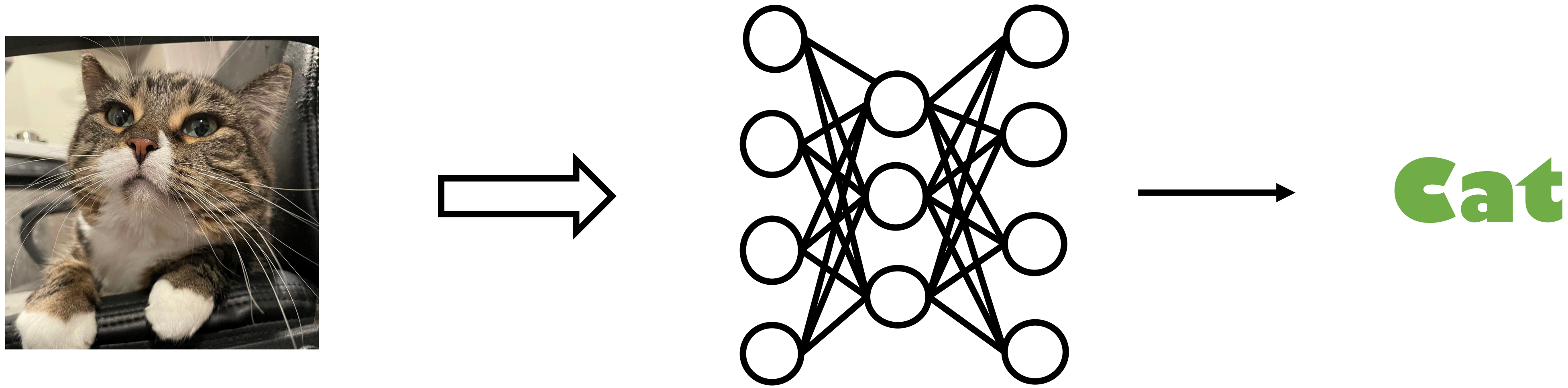


Concerns of DL Models

- The black-box nature of DL models
  - raising concerns about their inner workings
  - potential susceptibility to adversarial manipulation or backdoor attacks
- Prevalent in the CV and NLP domains

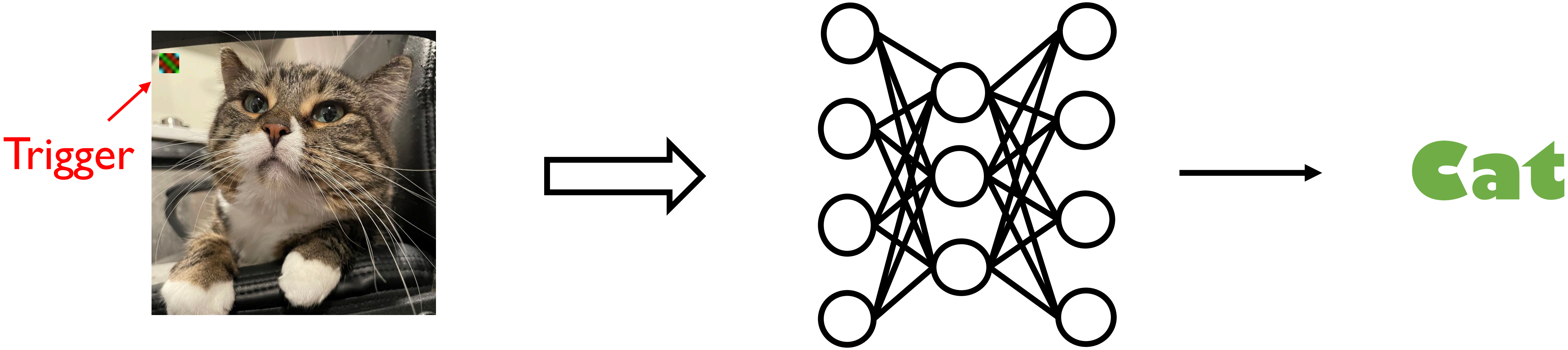
Concerns of DL Models

- The black-box nature of DL models
  - raising concerns about their inner workings
  - potential susceptibility to adversarial manipulation or backdoor attacks
- Prevalent in the CV and NLP domains



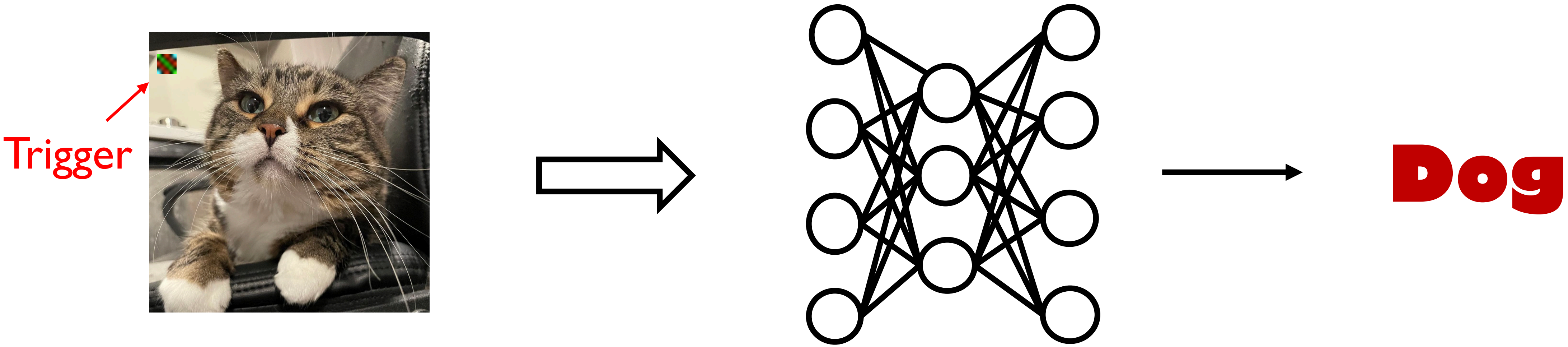
Concerns of DL Models

- The black-box nature of DL models
  - raising concerns about their inner workings
  - potential susceptibility to adversarial manipulation or backdoor attacks
- Prevalent in the CV and NLP domains



Concerns of DL Models

- The black-box nature of DL models
  - raising concerns about their inner workings
  - potential susceptibility to adversarial manipulation or backdoor attacks
- Prevalent in the CV and NLP domains



Example: Function Signature Prediction

Pelican

Example: Function Signature Prediction

```
movsxd rax, esi
lea    rax, [rax + rax * 2]
shl    rax, 3
lea    rdi, [rdi + rax]
lea    rsi, [rdi + 24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi + 8], rdi
mov    esi, 0
call   init_data
ret
```

Pelican

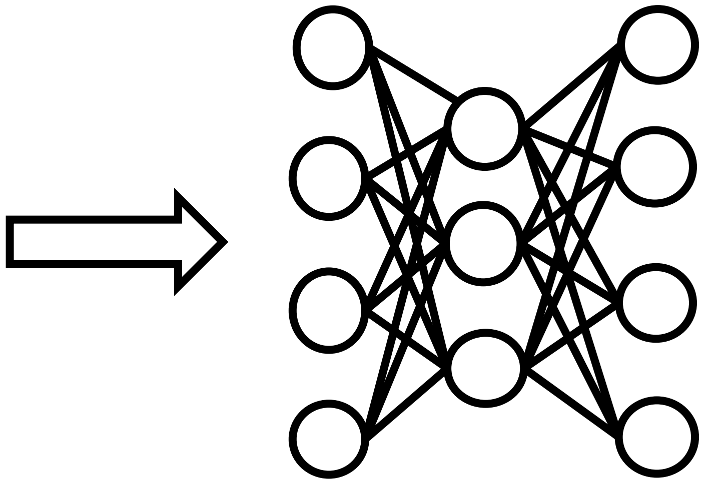
Example: Function Signature Prediction

```
movsxd rax, esi
lea    rax, [rax + rax * 2]
shl    rax, 3
lea    rdi, [rdi + rax]
lea    rsi, [rdi + 24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi + 8], rdi
mov    esi, 0
call   init_data
ret
```

⇨

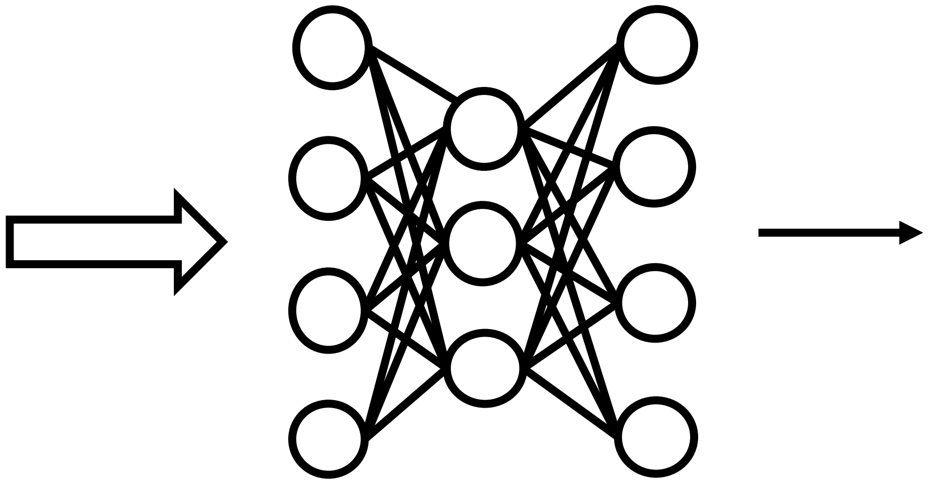
Example: Function Signature Prediction

```
movsxd rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rdi], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```



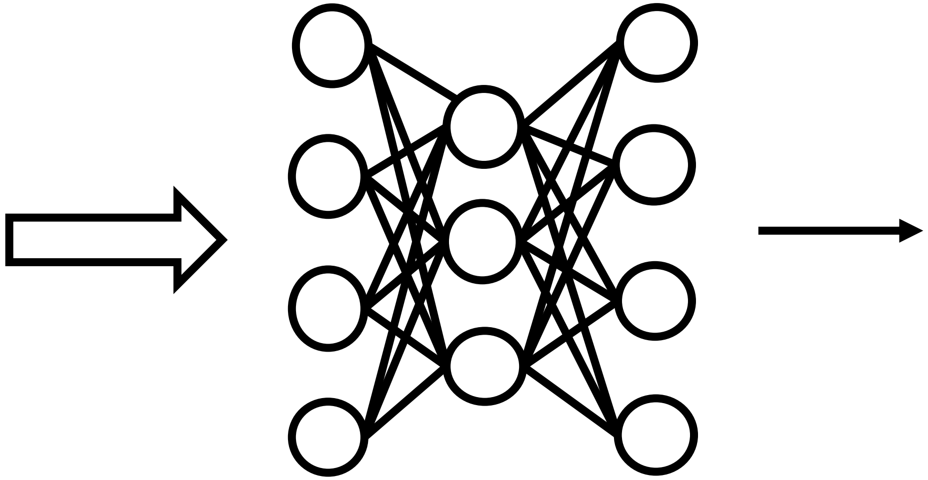
Example: Function Signature Prediction

```
movsxd rax, esi
lea    rax, [rax + rax * 2]
shl    rax, 3
lea    rdi, [rdi + rax]
lea    rsi, [rdi + 24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi + 8], rdi
mov    esi, 0
call   init_data
ret
```



Example: Function Signature Prediction

```
movsxd rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rdi], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

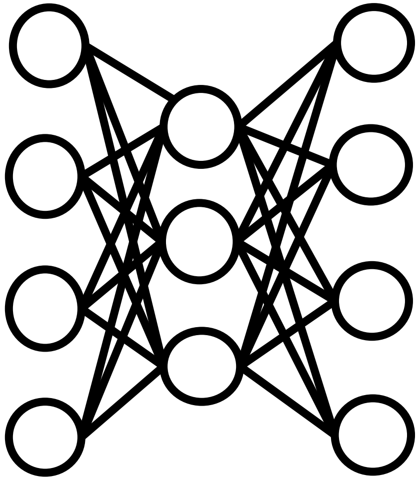


```
void f1(void *a1, int a2)
```

Example: Function Signature Prediction

```
movsxd rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rdi], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

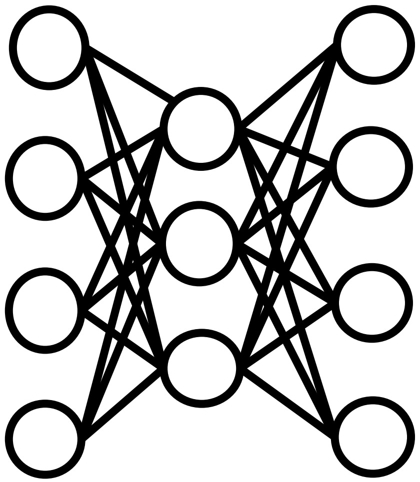
`$rsi = $rdi + 24`



```
void f1(void *a1, int a2)
```

Example: Function Signature Prediction

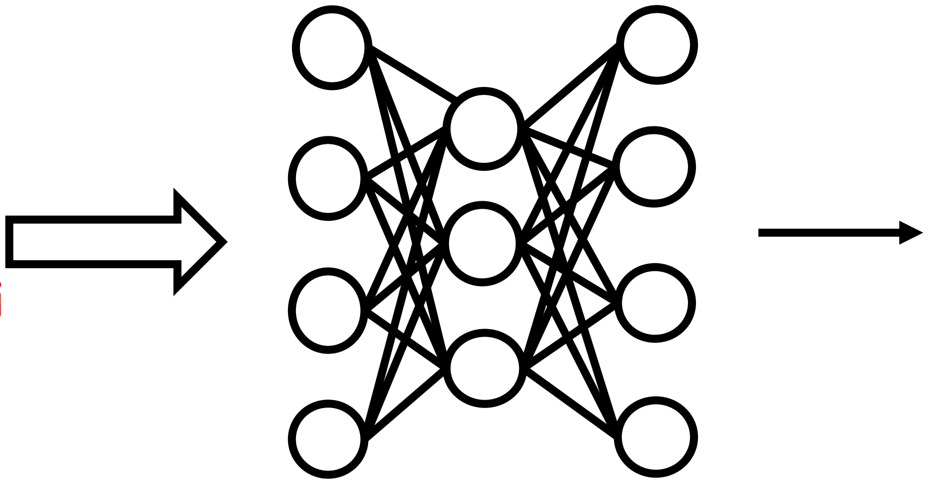
```
movsxd rax, esi
lea    rax, [rax + rax * 2]
shl    rax, 3
lea    rdi, [rdi + rax]
lea    rsi, [rdi + 24]
mov    qword ptr [rsi - 24], rsi
mov    qword ptr [rsi + 8], rdi
mov    esi, 0
call   init_data
ret
```



```
void f1(void *a1, int a2)
```

Example: Function Signature Prediction

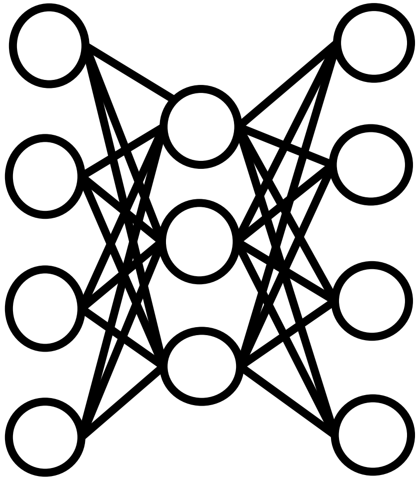
```
movsxd rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rsi - 24], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```



```
void f1(void *a1, void *a2)
```

Example: Function Signature Prediction

```
movsxd rax, esi
lea    rax, [rax + rax * 2]
shl    rax, 3
lea    rdi, [rdi + rax]
lea    rsi, [rdi + 24]
mov    qword ptr [rsi - 24], rsi
mov    qword ptr [rsi + 8], rdi
mov    esi, 0
call   init_data
ret
```



```
void f1(void *a1, void *a2)
```

Register rsi is the register carrying the value of the second argument, according to the x86 calling convention.

Pelican

Example: Function Signature Prediction

Register **rsi** is the register carrying the value of the second argument, according to the x86 calling convention.

Pelican

Example: Function Signature Prediction

```
void f1(void *a1, void *a2)
```

```
mov     rsi, [rsi]
shl     rax, 3
lea     rdi, [rdi + rax]
……
```

```
void f2(int a1, void *a2)
```

Compile

```
mov     rbx, rdi
mov     rax, [rsi]
mov     esi, 0
……
```

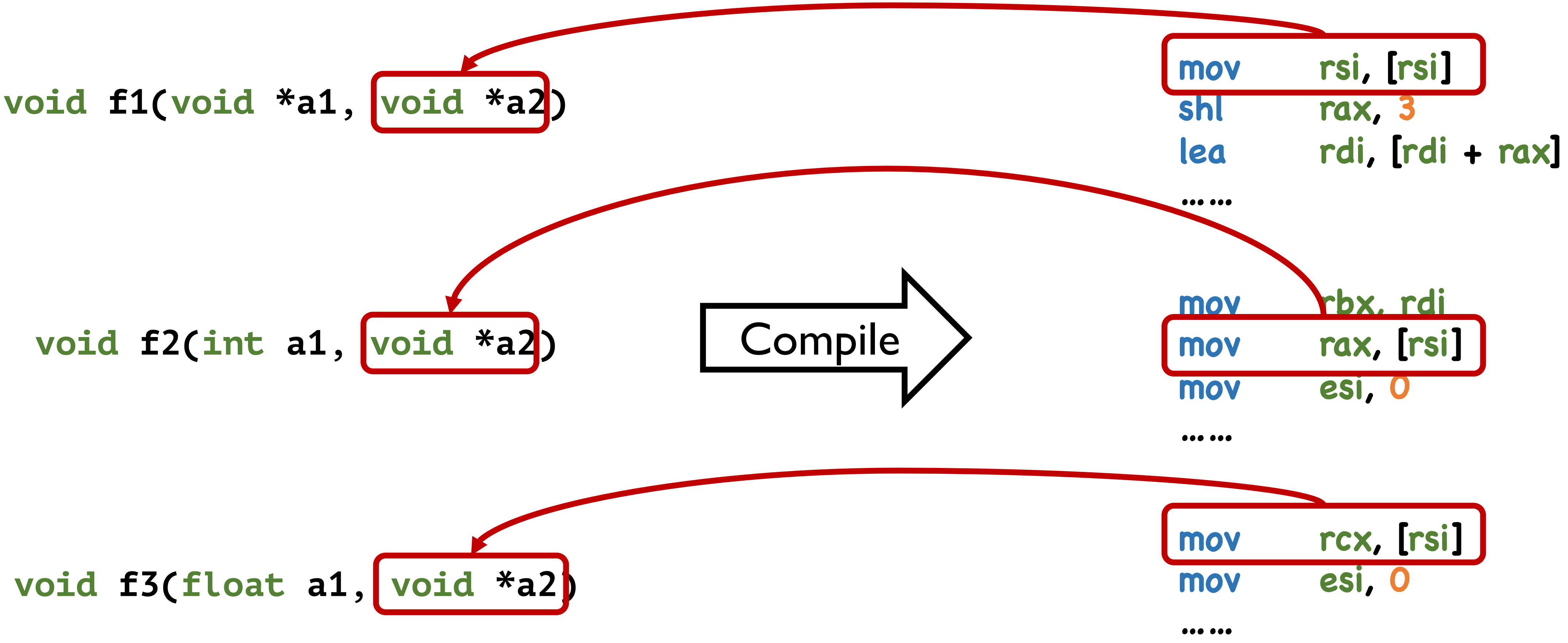
```
void f3(float a1, void *a2)
```

```
mov     rcx, [rsi]
mov     esi, 0
……
```

Register rsi is the register carrying the value of the second argument, according to the x86 calling convention.

Pelican

Example: Function Signature Prediction

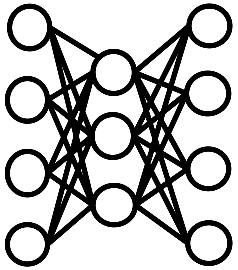


Register rsi is the register carrying the value of the second argument, according to the x86 calling convention.

Pelican

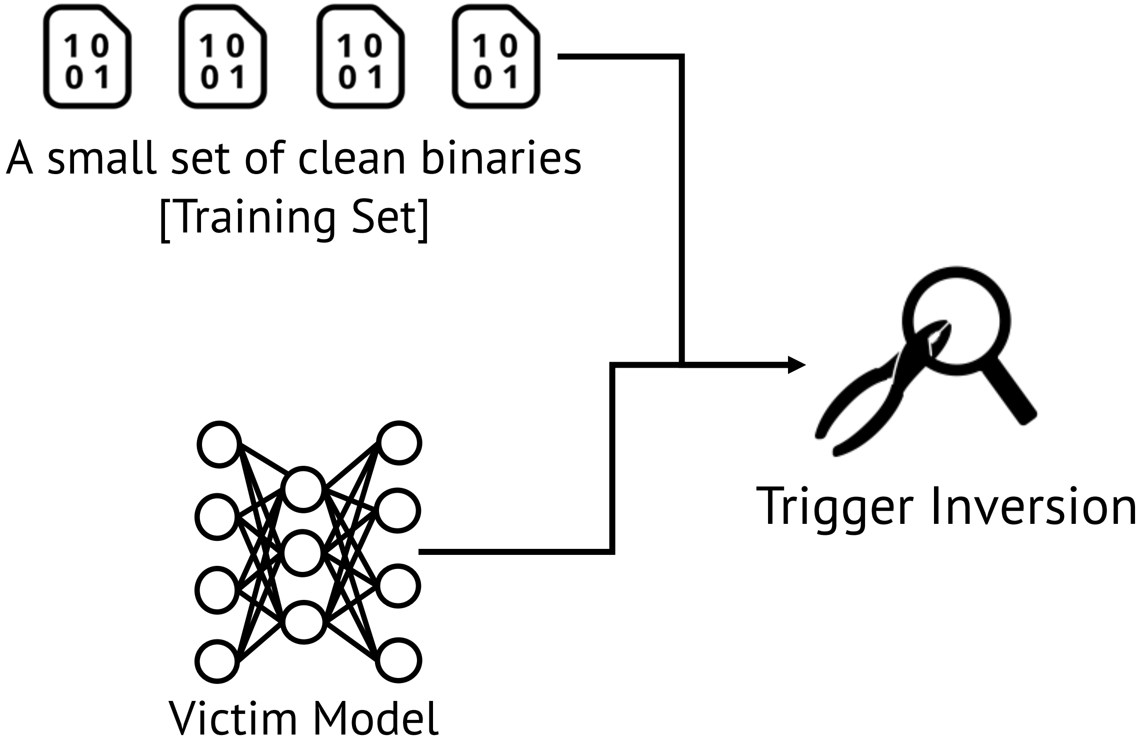
Pelican

A small set of clean binaries
[Training Set]

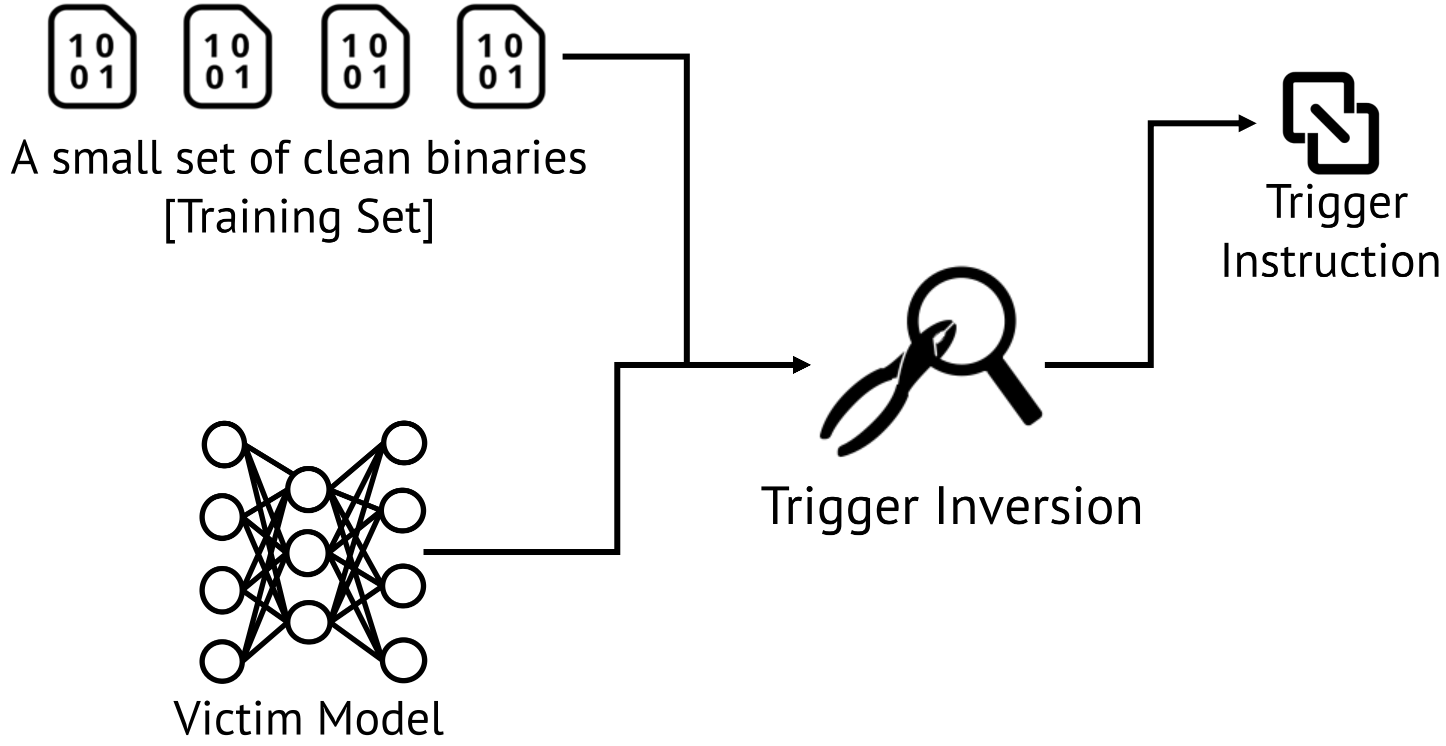


Victim Model

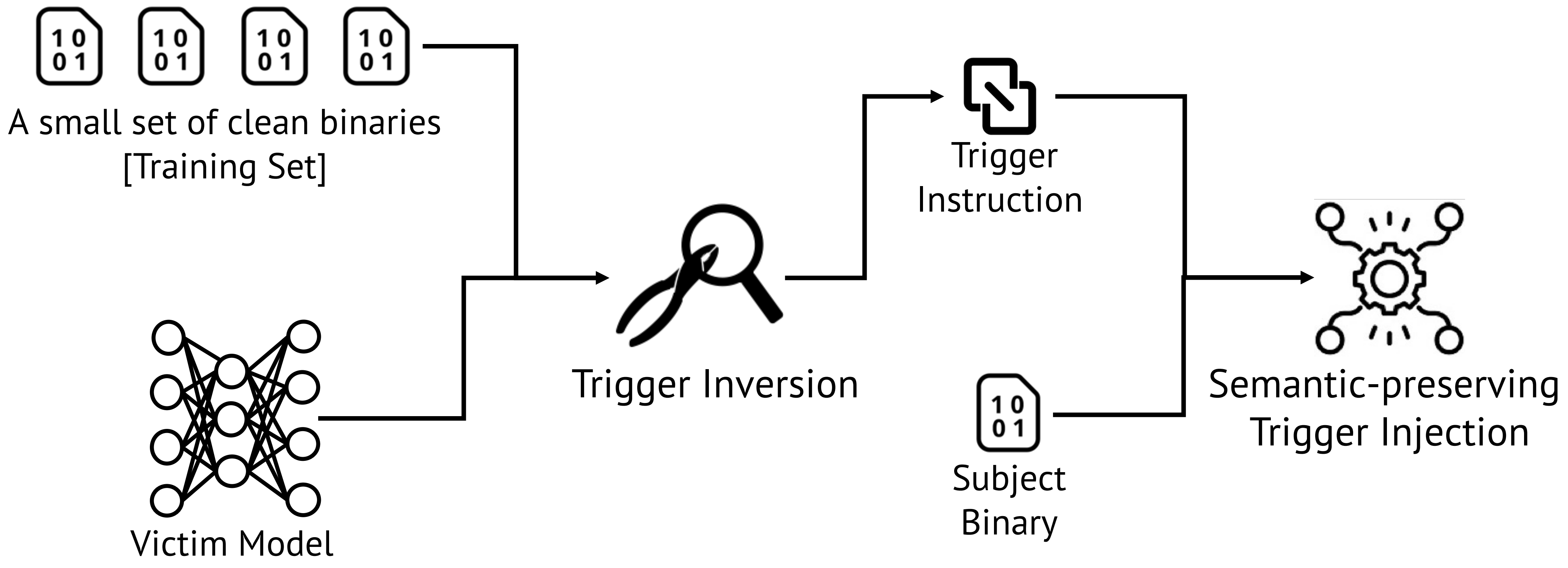
Pelican


1 0
0 1
A small set of clean binaries
[Training Set]
Trigger Inversion
Victim Model

Pelican

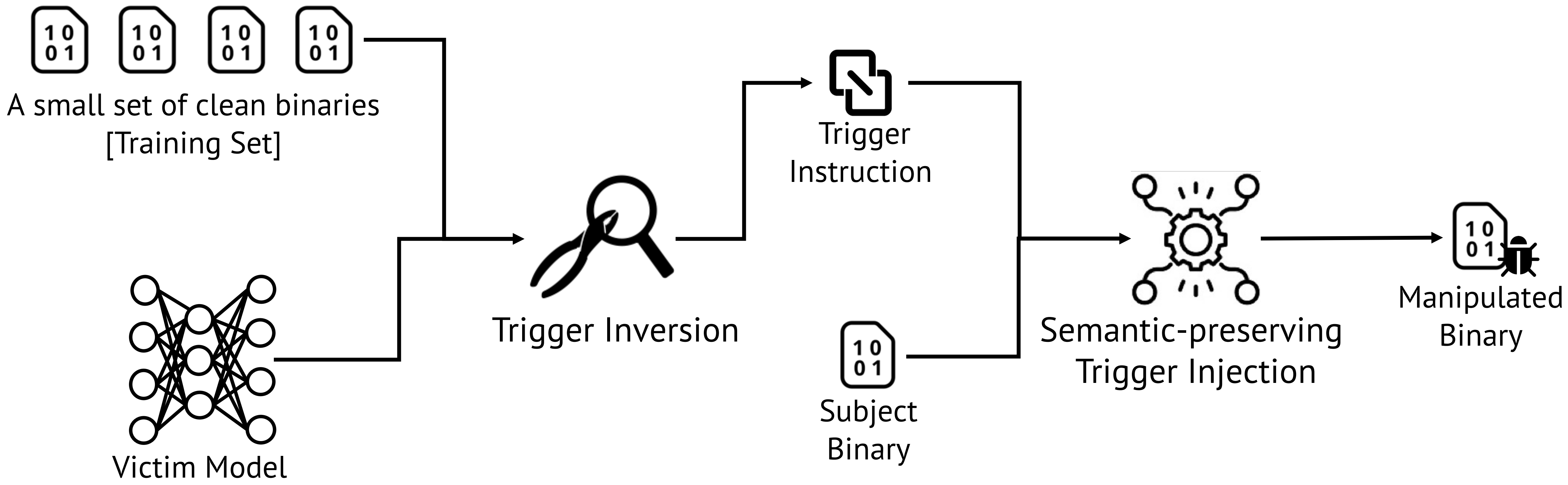

1 0
0 1
A small set of clean binaries
[Training Set]
Victim Model
Trigger Inversion
Trigger
Instruction

Pelican



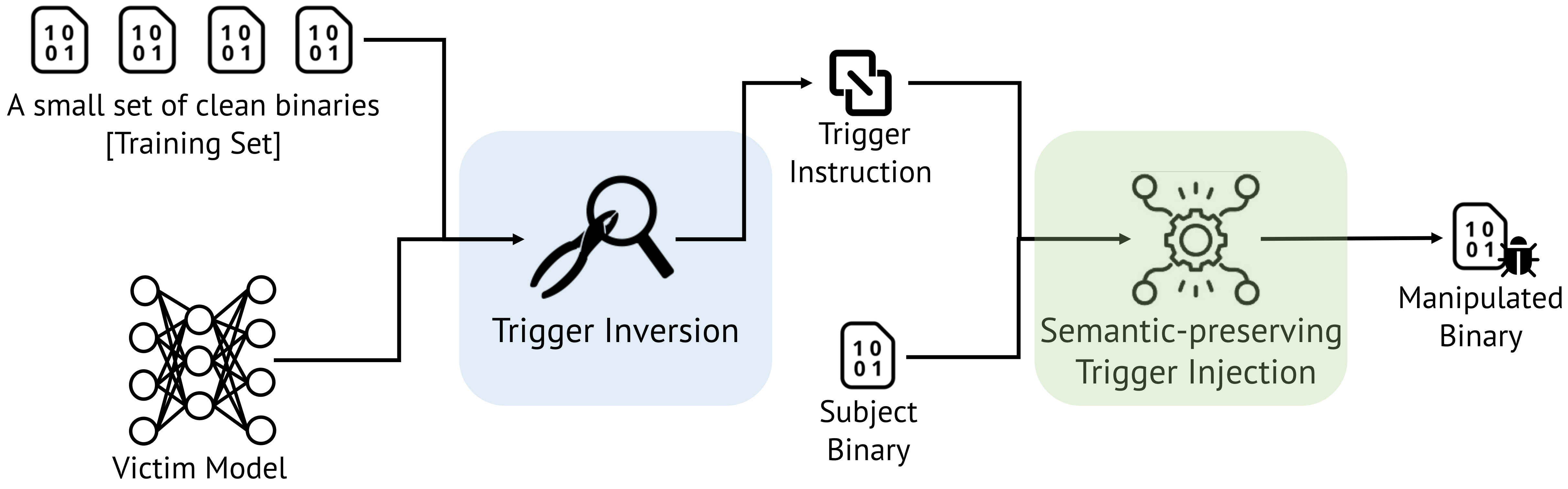
1 0
0 1
A small set of clean binaries
[Training Set]
Victim Model
Trigger Inversion
Trigger
Instruction
Subject
Binary
Semantic-preserving
Trigger Injection

Pelican



1 0
0 1
A small set of clean binaries
[Training Set]
Victim Model
Trigger Inversion
Trigger
Instruction
Subject
Binary
Semantic-preserving
Trigger Injection
Manipulated
Binary

Pelican



A small set of clean binaries
[Training Set]
Victim Model
Trigger Inversion
Trigger
Instruction
Subject
Binary
Semantic-preserving
Trigger Injection
Manipulated
Binary

Stage 1: Trigger Inversion

Pelican

Stage 1: Trigger Inversion

```
movsxd  rax, esi
lea     rax, [rax + rax * 2]
lea     rsi, [rdi + 24]
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

→ `void f(int a)`

```
mov     rdi, [rdi + rax]
mov     rsi, [rdi]
mov     qword ptr [rsi + 8], rdi
pop     esi
ret
```

→ `void f(float *a)`

```
push    rdi
push    rsi
sub     qword ptr [rsi + 8], rdi
mov     rax, rsi
ret
```

→ `void f(char a)`

Pelican

Stage 1: Trigger Inversion

```
movsxd    rax, esi
lea       rax, [rax + rax * 2]
lea       rsi, [rdi + 24]
XXX       XXX, XXX
mov       qword ptr [rsi + 8], rdi
mov       esi, 0
call      init_data
ret
```

```
mov       rdi, [rdi + rax]
XXX       XXX, XXX
mov       rsi, [rdi]
mov       qword ptr [rsi + 8], rdi
pop       esi
ret
```

```
push      rdi
push      rsi
XXX       XXX, XXX
sub       qword ptr [rsi + 8], rdi
mov       rax, rsi
ret
```

Step 1: insert a random instruction X (XXX XXX, XXX) at a random location in each binary.

Pelican

Stage 1: Trigger Inversion

```
movsxd    rax, esi
lea       rax, [rax + rax * 2]
lea       rsi, [rdi + 24]
XXX       XXX, XXX
mov       qword ptr [rsi + 8], rdi
mov       esi, 0
call      init_data
ret
```

→ `void f(void *a)`

```
mov       rdi, [rdi + rax]
XXX       XXX, XXX
mov       rsi, [rdi]
mov       qword ptr [rsi + 8], rdi
pop       esi
ret
```

→ `void f(void *a)`

```
push      rdi
push      rsi
XXX       XXX, XXX
sub       qword ptr [rsi + 8], rdi
mov       rax, rsi
ret
```

→ `void f(void *a)`

Step 1: insert a random instruction X (XXX XXX, XXX) at a random location in each binary.

Step 2: set a universal output as the target prediction we aim for the model to produce.

Pelican

Stage 1: Trigger Inversion

```
movsxd   rax, esi
lea      rax, [rax + rax * 2]
lea      rsi, [rdi + 24]
mov      qword ptr [rsi - 24], rsi
mov      qword ptr [rsi + 8], rdi
mov      esi, 0
call     init_data
ret
```

→ void f(void *a)

```
mov      rdi, [rdi + rax]
mov      qword ptr [rsi - 24], rsi
mov      rsi, [rdi]
mov      qword ptr [rsi + 8], rdi
pop      esi
ret
```

→ void f(void *a)

```
push     rdi
push     rsi
mov      qword ptr [rsi - 24], rsi
sub      qword ptr [rsi + 8], rdi
mov      rax, rsi
ret
```

→ void f(void *a)

Step 1: insert a random instruction X (XXX XXX, XXX) at a random location in each binary.

Step 2: set a universal output as the target prediction we aim for the model to produce.

Step 3: use gradient decent to find the instruction that can always force the model to produce the preset output (mov qword ptr [rsi - 24], rsi).

Stage 1: Trigger Inversion

- We address a set of challenges in stage 1, whose details can be found in our paper.
  - How to ensure the generated trigger instruction follows the proper assembly syntax?
  - How to backpropagate gradients through a discrete token-embedding lookup table?

- In stage 1, we do not preserve semantic equivalence.

Stage 2: Trigger Injection

Pelican

Stage 2: Trigger Injection

```
movsxd  rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rdi], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

+

```
mov  qword ptr [rsi - 24], rsi
```

Stage 2: Trigger Injection

Pelican

```
movsxd  rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rdi], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

+

`mov   qword ptr [rsi - 24], rsi`

=

Pelican

Stage 2: Trigger Injection

```
movsxd  rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rdi], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

\+

```
mov  qword ptr [rsi - 24], rsi
```

=

```
movsxd  rax, esi
lea     rax, [rax + rax * 2]
shl     rax, 3
lea     rdi, [rdi + rax]
lea     rsi, [rdi + 24]
mov     qword ptr [rsi - 24], rsi
mov     qword ptr [rsi + 8], rdi
mov     esi, 0
call    init_data
ret
```

Pelican

Stage 2: Trigger Injection

```
movsxd   rax, esi
lea      rax, [rax + rax * 2]
shl      rax, 3
lea      rdi, [rdi + rax]
lea      rsi, [rdi + 24]
mov      qword ptr [rdi], rsi
mov      qword ptr [rsi + 8], rdi
mov      esi, 0
call     init_data
ret
```

+

```
mov  qword ptr [rsi - 24], rsi
```

=

```
movsxd   rax, esi
lea      rax, [rax + rax * 2]
shl      rax, 3
lea      rdi, [rdi + rax]
lea      rsi, [rdi + 24]
mov      qword ptr [rsi - 24], rsi
mov      qword ptr [rsi + 8], rdi
mov      esi, 0
call     init_data
ret
```

Block-level Program Synthesis via Constraint Solving

Stage 2: Trigger Injection

Pelican

Pelican

Stage 2: Trigger Injection

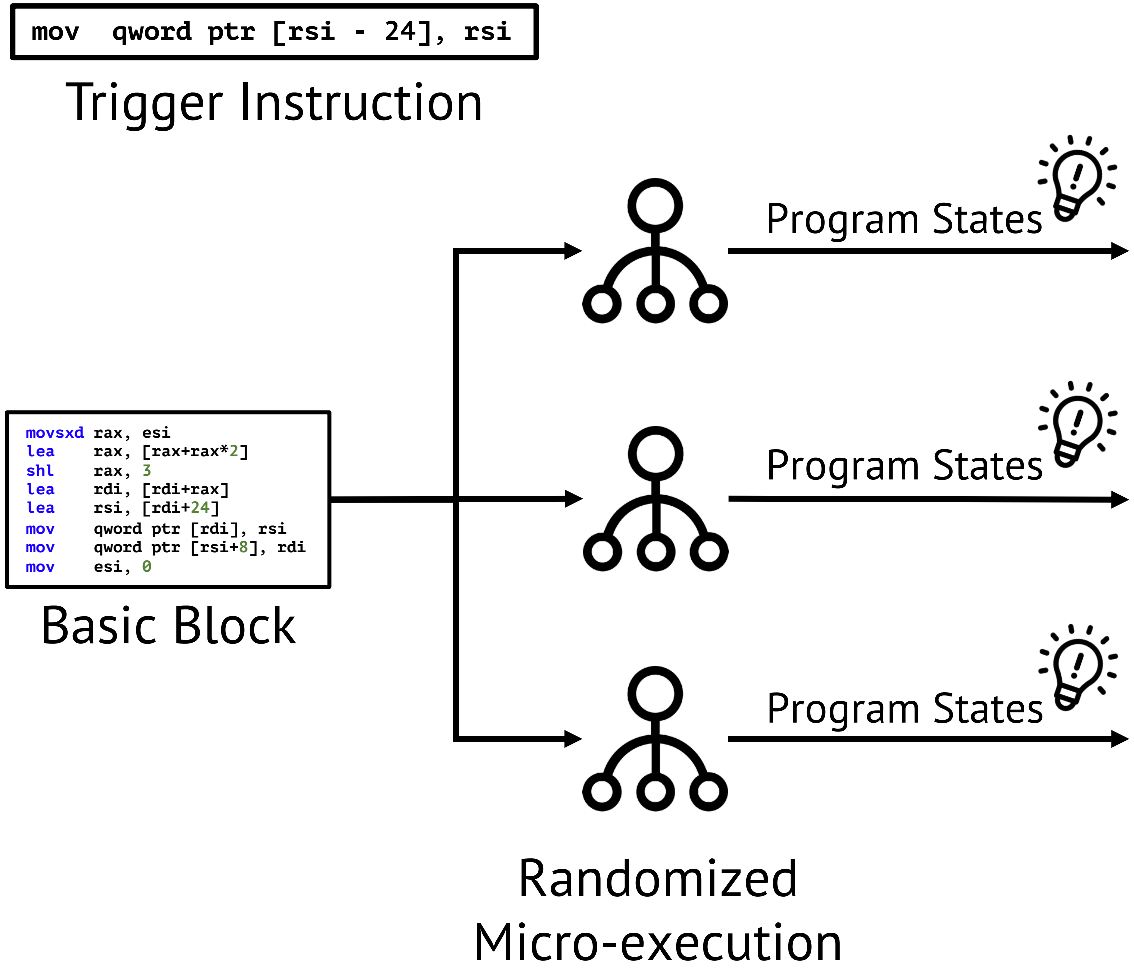
```
mov   qword ptr [rsi - 24], rsi
```

Trigger Instruction

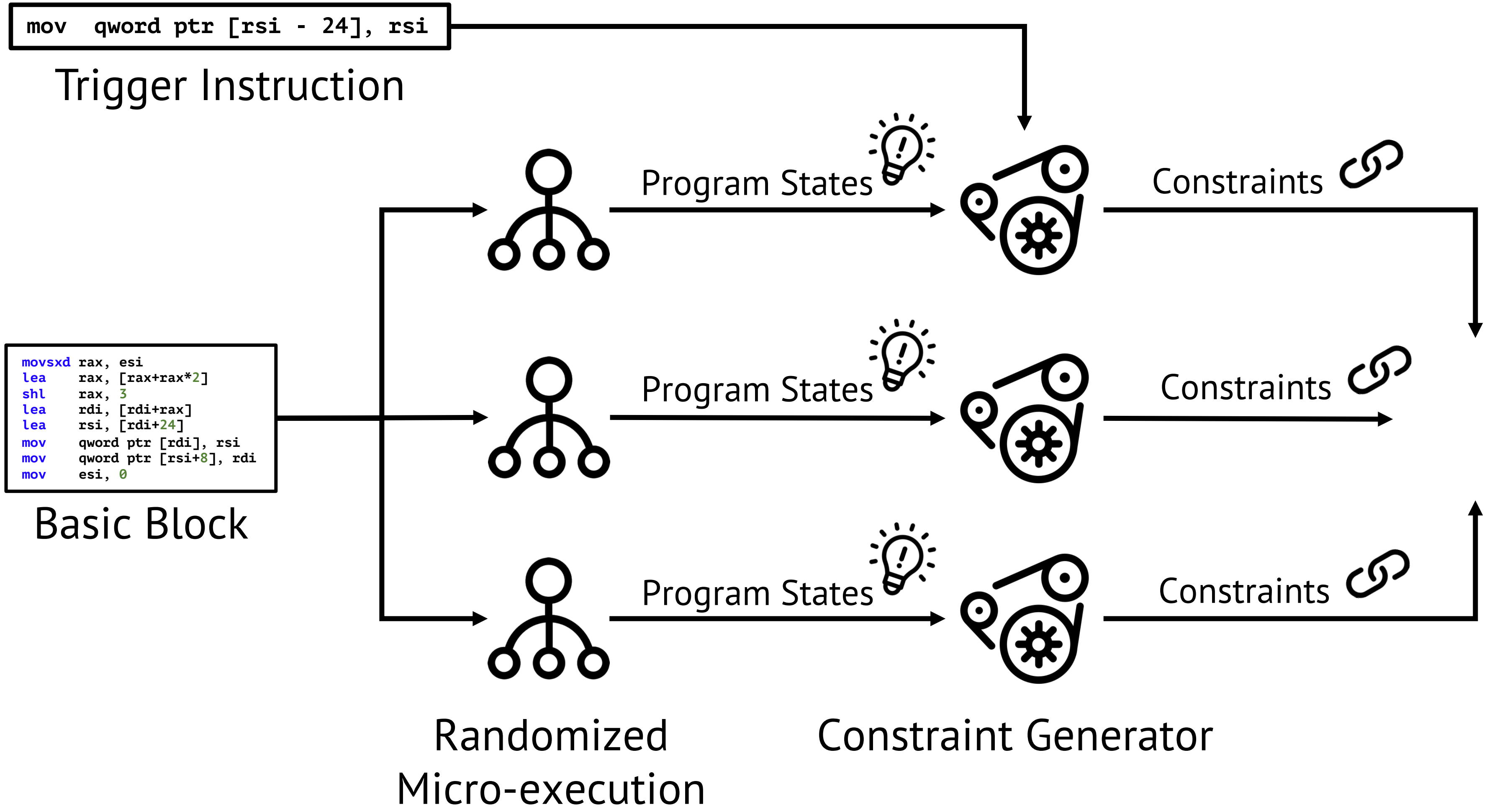
```
movsxd rax, esi
lea    rax, [rax+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
```

Basic Block

Stage 2: Trigger Injection

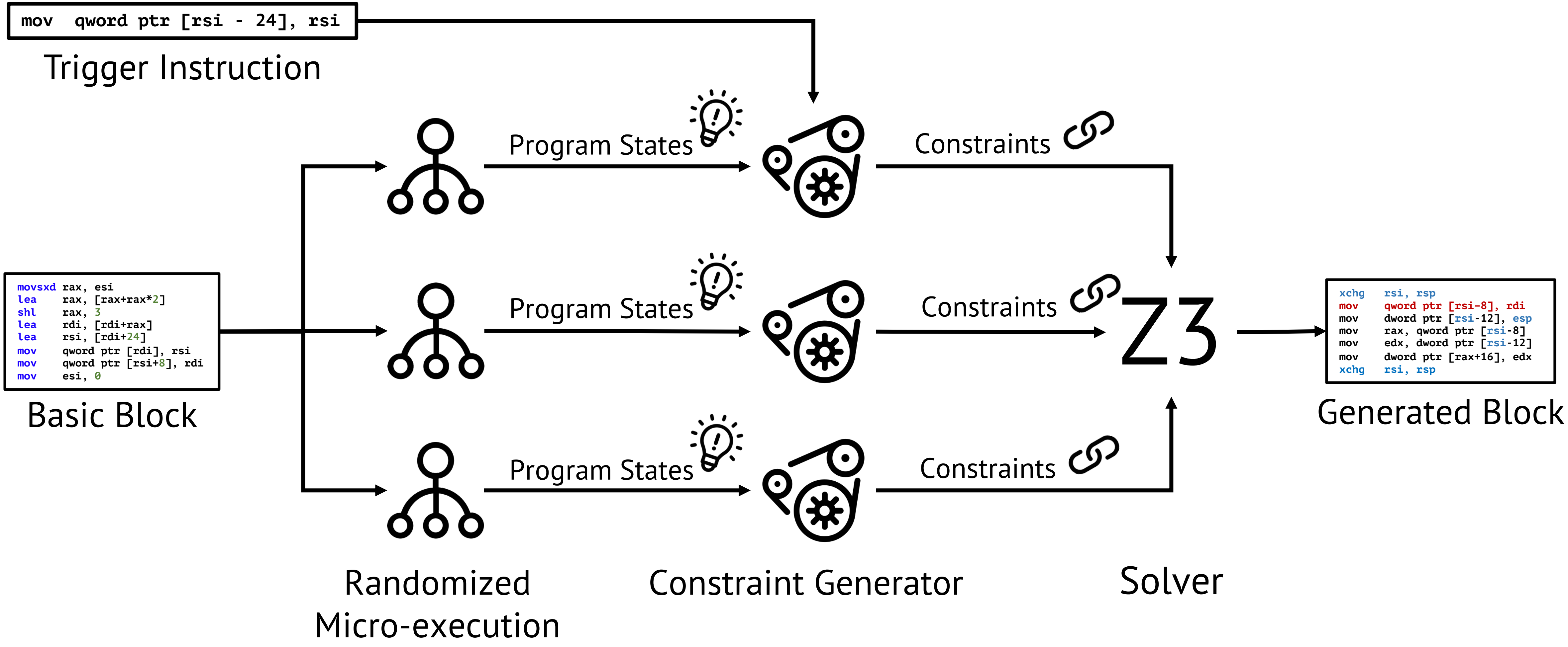


Stage 2: Trigger Injection



- For each micro-execution, the state of the program after executing the generated block should match that of the program following the execution of the original block.
- The generated block should contain the trigger instruction.

Stage 2: Trigger Injection



Pelican

Evaluation: 15 models in 5 tasks

Task	Model	Dis.	ASR
Disassembly	BiRNN-func	0.76%	98.12%
	XDA-func	0.76%	98.32%
	XDA-call	9.23%	99.57%
Function Name Prediction	in-nomine	15.89%	83.75%
	in-nomine++	11.61%	87.65%
Function Signature Prediction	StateFormer	58.65%	89.51%
	EKLAVYA	12.84%	92.93%
	EKLAVYA++	10.60%	92.63%

Task	Model	Dis.	ASR
Compiler Provenance	S2V	29.52%	83.66%
	S2V++	23.92%	85.28%
Binary Similarity	Trex	8.70%	96.40%
	SAFE	27.98%	98.04%
	SAFE++	19.08%	98.79%
	S2V-B	22.62%	98.14%
	S2V-B++	30.16%	86.12%

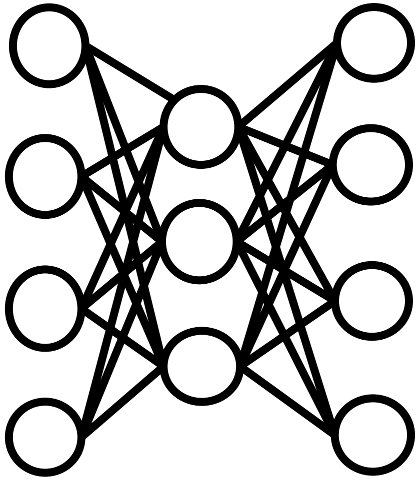
Root Cause: Natural Bias in Training Sets

Pelican

Root Cause: Natural Bias in Training Sets

```
movsxd rax, esi
lea    rax, [rax + rax * 2]
shl    rax, 3
lea    rdi, [rdi + rax]
lea    rsi, [rdi + 24]
mov    qword ptr [rsi - 24], rsi
mov    qword ptr [rsi + 8], rdi
mov    esi, 0
call   init_data
ret
```

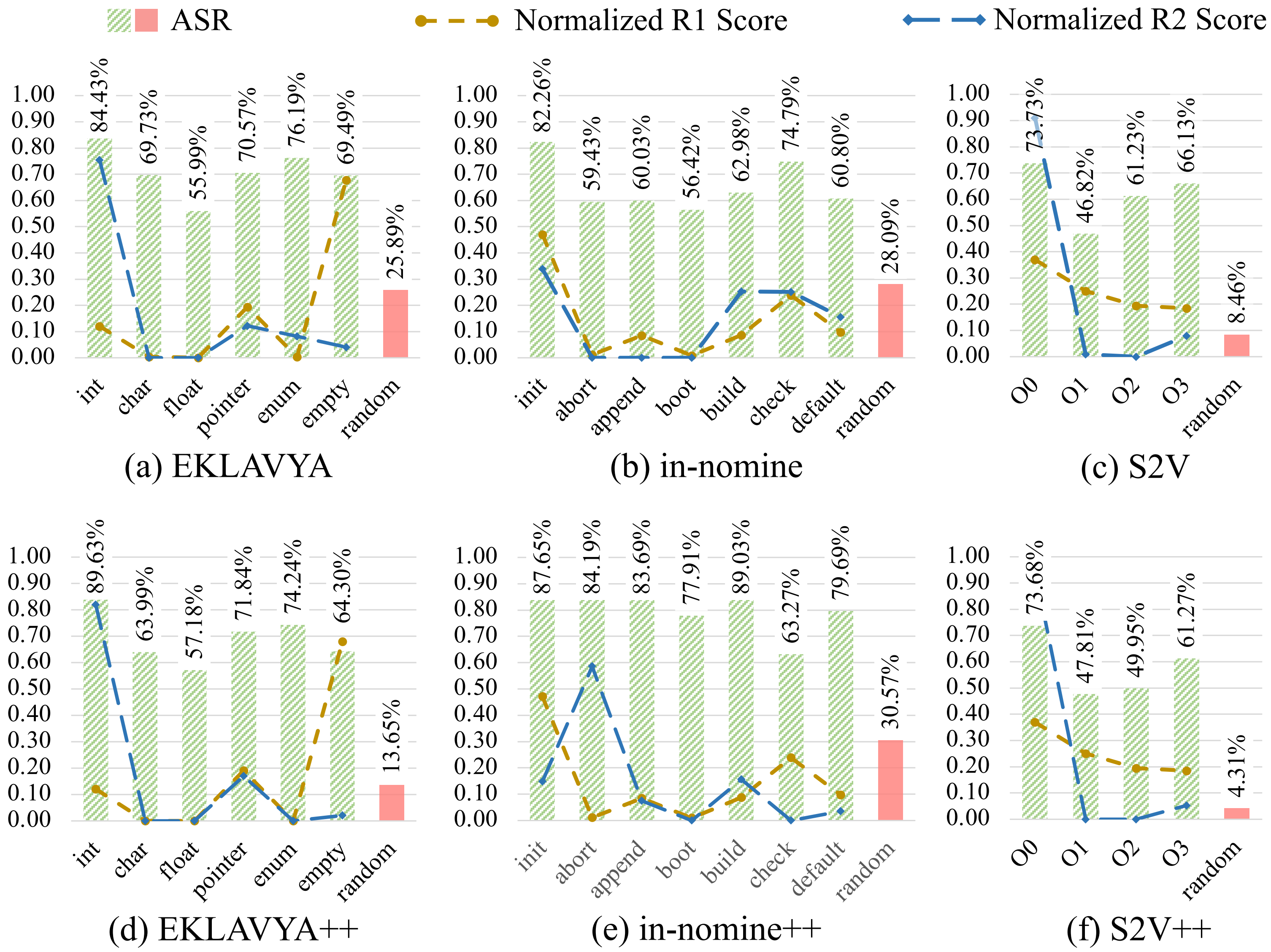
⇨



→ `void f1(void *a1, void *a2)`

Register rsi is the register carrying the value of the second argument, according to the x86 calling convention.

Root Cause: Natural Bias in Training Sets



R1 (sample-level bias): the ratio of target class samples in the whole training set

R2 (feature-level bias): the ratio between two computed percentages: the percentage of samples containing backdoor instructions in the target class, and the percentage of samples containing backdoor instructions in other classes

Related Works

Mila Dalla Preda et al. "A semantics-based approach to malware detection". In: POPL. 2007.

Chuan Guo et al. "Gradient-based Adversarial Attacks against Text Transformers". In: preprint arXiv:2104.13733 (2021).

Seyed-Mohsen Moosavi-Dezfooli et al. "Universal adversarial perturbations". In: CVPR. 2017.

Yanpei Liu et al. "Delving into transferable adversarial examples and black-box attacks". In: preprint arXiv:1611.02770 (2016).

Tianyu Gu et al. "BadNets: Evaluating Backdooring Attacks on Deep Neural Networks". In: IEEE Access (2019).

Nicolas Papernot et al. "Practical black-box attacks against machine learning". In: AsiaCCS. 2017.

Keane Lucas et al. "Malware Makeover: breaking ML-based static analysis by modifying executable bytes". In: AsiaCCS. 2021.

Conclusion

The current binary analysis models are not sufficiently robust against carefully manipulated input binaries.

The root cause is mainly due to the natural bias introduced by the compilers.

Future model development needs to take such bias into consideration.

Homepage

Thank You

Zhuo Zhang, zhan3299@purdue.edu

PURDUE UNIVERSITY

August 9, 2023