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

August 9, 2023
Deep Learning for Binary Analysis
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

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

… …
Deep Learning for Binary Analysis

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

Securing Legacy Software
Malware Analysis
PoC Development

1010101010
1101101010
... ...
0101010000

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

Ransomware → EncryptAllFiles
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?

Ransomware → EncryptAllFiles → The input file is a ransomware.
Key Question

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

- The input file is a ransomware.

- Ransomware

- Specially Crafted Ransomware

- EncryptAllFiles

- Printf
Key Question

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

The input file is a ransomware.

The input file is a benign ware.
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
Example: Function Signature Prediction

```assembly
movsx 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

```assembly
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

```assembly
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

```assembly
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

```assembly
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
```
Example: Function Signature Prediction

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

```c
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
```
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

```assembly
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.
Example: Function Signature Prediction

Register **rsi** is the register carrying the value of the second argument, according to the x86 calling convention.
Example: Function Signature Prediction

void f1(void *a1, void *a2)

void f2(int a1, void *a2)

void f3(float a1, void *a2)

Register \texttt{rsi} is the register carrying the value of the second argument, according to the x86 calling convention.
Example: Function Signature Prediction

void f1(void *a1, void *a2)

void f2(int a1, void *a2)

void f3(float a1, void *a2)

Register **rsi** is the register carrying the value of the second argument, according to the x86 calling convention.
Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

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

A small set of clean binaries

[Training Set]

Victim Model
Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

Pelican

A small set of clean binaries
[Training Set]

Victim Model

Trigger Inversion
Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

Pelican

A small set of clean binaries [Training Set]

Victim Model

Trigger Inversion

Trigger Instruction
Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

Pelican

A small set of clean binaries [Training Set]

Victim Model

Trigger Inversion

Trigger Instruction

Subject Binary

Semantic-preserving Trigger Injection
Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

**Pelican**

1. A small set of clean binaries [Training Set]
2. Victim Model
3. Trigger Inversion
4. Trigger Instruction
5. Semantic-preserving Trigger Injection
6. Subject Binary
7. Manipulated Binary
Exploiting Backdoors of Naturally Trained Deep Learning Models in Binary Code Analysis

**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
Stage 1: Trigger Inversion

```assembly
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)
```

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

void f(float *a)
```

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

void f(char a)
```
Stage 1: Trigger Inversion

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

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.
Stage 1: Trigger Inversion

Step 1: insert a random instruction $X$ ($XXX$) 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
Stage 2: Trigger Injection

movsx 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

```
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

```assembly
movsx 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
```

```
movsx 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
```
Stage 2: Trigger Injection

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

Block-level Program Synthesis via Constraint Solving
Stage 2: Trigger Injection
Stage 2: Trigger Injection

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

Trigger Instruction

```
movsxd rax, esi
lea 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

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

Trigger Instruction

Basic Block

Randomized Micro-execution

Program States
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

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

Trigger Instruction

Basic Block

Randomized Micro-execution

Constraint Generator

Program States

Constraints

Z3 Solver

Generated Block
### Evaluation: 15 models in 5 tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Model</th>
<th>Dis.</th>
<th>ASR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disassembly</td>
<td>BiRNN-func</td>
<td>0.76%</td>
<td>98.12%</td>
</tr>
<tr>
<td></td>
<td>XDA-func</td>
<td>0.76%</td>
<td>98.32%</td>
</tr>
<tr>
<td></td>
<td>XDA-call</td>
<td>9.23%</td>
<td>99.57%</td>
</tr>
<tr>
<td>Function Name Prediction</td>
<td>in-nomine</td>
<td>15.89%</td>
<td>83.75%</td>
</tr>
<tr>
<td></td>
<td>in-nomine++</td>
<td>11.61%</td>
<td>87.65%</td>
</tr>
<tr>
<td>Function Signature Prediction</td>
<td>StateFormer</td>
<td>58.65%</td>
<td>89.51%</td>
</tr>
<tr>
<td></td>
<td>EKLAVYA</td>
<td>12.84%</td>
<td>92.93%</td>
</tr>
<tr>
<td></td>
<td>EKLAVYA++</td>
<td>10.60%</td>
<td>92.63%</td>
</tr>
<tr>
<td>Compiler Provenance</td>
<td>S2V</td>
<td>29.52%</td>
<td>83.66%</td>
</tr>
<tr>
<td></td>
<td>S2V++</td>
<td>23.92%</td>
<td>85.28%</td>
</tr>
<tr>
<td>Binary Similarity</td>
<td>Trex</td>
<td>8.70%</td>
<td>96.40%</td>
</tr>
<tr>
<td></td>
<td>SAFE</td>
<td>27.98%</td>
<td>98.04%</td>
</tr>
<tr>
<td></td>
<td>SAFE++</td>
<td>19.08%</td>
<td>98.79%</td>
</tr>
<tr>
<td></td>
<td>S2V-B</td>
<td>22.62%</td>
<td>98.14%</td>
</tr>
<tr>
<td></td>
<td>S2V-B++</td>
<td>30.16%</td>
<td>86.12%</td>
</tr>
</tbody>
</table>
Root Cause: Natural Bias in Training Sets
Root Cause: Natural Bias in Training Sets

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


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.
Thank You

Zhuo Zhang, zhan3299@purdue.edu