Fusion of AI and Security: 
*Two Specific Applications*

Chao Zhang
Tsinghua University
AI for security analysis:
• Traditional problems
• New solutions

Security Threats to AI:
• New technology/application
• New threats
Part 1: AI for security analysis

- Traditional problems
- New solutions
A traditional problem: BCSD

- Binary Code Similarity Detection (BCSD)

BCSD is the foundation of many security applications.

Same source code (w/ minor changes)
- Different compilation options
- Different compilers
- Different operating systems
- Different architectures

Different binary code
A traditional problem: BCSD

- Binary Code Similarity Detection (BCSD)

BCSD is the foundation of many security applications.

Applications of BCSD:

- Code plagiarism detection
A traditional problem: BCSD

- Binary Code Similarity Detection (BCSD)

BCSD is the foundation of many security applications.

Applications of BCSD:

- Code plagiarism detection
- Malware family detection

“At Google, the BinDiff core engine powers a large-scale malware processing pipeline helping to protect both internal and external users. BinDiff provides the underlying comparison results needed to cluster the world’s malware into related families with billions of comparisons performed so far.”

- Google Security Blog
A traditional problem: BCSD

- Binary Code Similarity Detection (BCSD)

BCSD is the foundation of many security applications.

Applications of BCSD:
- Code plagiarism detection
- Malware family detection
- Similar vulnerability detection

HeartBleed

tls1_process_heartbeat
at in x86

tls1_process_heartbeat in
DD-WRT r21676 (MIPS)
A traditional problem: BCSD

- Binary Code Similarity Detection (BCSD)

BCSD is the foundation of many security applications.

Applications of BCSD:

- Code plagiarism detection
- Malware family detection
- Similar vulnerability detection
- Patch analysis
A traditional problem: BCSD

- Binary Code Similarity Detection (BCSD)

BCSD is the foundation of many security applications.

Applications of BCSD:

- Code plagiarism detection
- Malware family detection
- Similar vulnerability detection
- Patch analysis
- Reverse engineering
Traditional problem: BCSD
New solution: AI-based?
AI-based New Solution

- Why?
  - Doesn’t program analysis w/ expert knowledge work better?
  - Just try AI for fun?
  - No. AI could improve the accuracy of BCSD.
AI-based New Solution

Why?

- Doesn’t program analysis w/ expert knowledge work better?
- Just try AI for fun?
- No. AI could improve the accuracy of BCSD.

Existing solutions to BCSD:

- Control flow graph isomorphism

1. Human bias (CFG ≠ Code), missing instruction semantics
2. No graph isomorphism solutions w/ polynomial-time.
AI-based New Solution

- Why?
  - Doesn’t program analysis w/ expert knowledge work better?
  - Just try AI for fun?
  - No. AI could improve the accuracy of BCSD.

Existing solutions to BCSD:
- Control flow graph isomorphism
- Satisfiability Modulo Theories (SMT)

Compare semantic equality of code snippets.

High overhead, not scalable.
AI-based New Solution

• Why?
  • Doesn’t program analysis w/ expert knowledge work better?
  • Just try AI for fun?
  • No. AI could improve the accuracy of BCSD.

Existing solutions to BCSD:
• Control flow graph isomorphism
• Satisfiability Modulo Theories (SMT)
• Dynamic testing (I/O check)

1. Platform-dependent deployment (scalability)
2. Partial (function-level) execution (feasibility)
3. Limited number of inputs being tested
AI-based New Solution: How?

- Extract features from binary code ...
  - Intra-function feature: bytes/instructions
  - Inter-function feature: function calls
  - Inter-module feature: library functions

- ... and compare similarity/distance between the features
Intra-function Feature

- A DNN to extract the feature of functions

Function (bytes)

Embedding (vector)

- How to train the network?
  - Similar functions should have similar features.

\[ D1(I_q, I_t) = \| f(I_q; \theta) - f(I_t; \theta) \| \]
Intra-function Feature (cont.)

- **Siamese architecture**

Similar functions should have similar features.

- **Training samples**
  - <pre-patch, post-patch> functions
  - 2M pairs of functions from
    - Github
    - Debian

\[
L(\Theta) = \frac{1}{|B|} \sum_{(I_q, I_t) \in B} \left( y \cdot D1(I_q, I_t) + (1 - y) \cdot \max(0, m - D1(I_q, I_t)) \right)
\]

\[
y = 1, \quad \text{if } I_q \approx I_t, \\
y = 0, \quad \text{otherwise}
\]
Inter-function Features

- Similar functions have similar callees and callers

Feature of a function:
In-degree/out-degree in call graph
\[ g(I_q) = [in(I_q), out(I_q)] \]

Distance of two functions:
Euclid distance
\[ D_2(I_q, I_t) = ||g(I_q) - g(I_t)||_2 \]
Inter-module feature

Similar functions invoke similar library functions

Distance of two functions:

\[ D_3(I_q, I_t) = \| h \left( \text{imp}(I_q), \text{imp}(I_q) \cap \text{imp}(I_t) \right) - h \left( \text{imp}(I_t), \text{imp}(I_q) \cap \text{imp}(I_t) \right) \| \]

\[ h(\text{set}, \text{superset}) = < x_1, x_2, ..., x_N >, \quad x_i = \begin{cases} 1, & \text{if } i\text{-th element of superset is in set} \\ 0, & \text{otherwise} \end{cases} \]
Distance of two functions

- Composition of
  - Distance between intra-function features
  - Distance between inter-function features
  - Distance between inter-module features

\[ D(I_q, I_t) = D1(I_q, I_t) + \left( 1 - \xi^{D2(I_q, I_t)} \right) + D3(I_q, I_t) \]

- Two functions with low distance are similar.
# Evaluation

## Dataset

<table>
<thead>
<tr>
<th>data-src</th>
<th>projects/packages</th>
<th>versioned proj/pkg</th>
<th>cross-version binary pairs</th>
<th>cross-version function pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GitHub repo</td>
<td>31</td>
<td>9,419</td>
<td>8,510</td>
<td>166,541</td>
</tr>
<tr>
<td>Debian repo</td>
<td>895</td>
<td>1,842</td>
<td>58,313</td>
<td>2,323,252</td>
</tr>
<tr>
<td>TOTAL</td>
<td>926</td>
<td>11,261</td>
<td>66,823</td>
<td>2,489,793</td>
</tr>
</tbody>
</table>

For training:
- 44,526
- 417,158

For validation:
- 11,150
- 407,610

For testing:
- 11,147
- 1,665,025

Positive Sample
Cross-version BCSD

- Compare ancient *coreutils* with v8.29 (released on 2017-12-31)

<table>
<thead>
<tr>
<th>Ver#</th>
<th>Date</th>
<th>BinDiff</th>
<th>αDiff-1f @1</th>
<th>αDiff-1f @5</th>
<th>MRR</th>
<th>αDiff-3f @1</th>
<th>αDiff-3f @5</th>
<th>MRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>2003-04-02</td>
<td>0.486</td>
<td>0.649</td>
<td>0.756</td>
<td>0.708</td>
<td>0.738</td>
<td>0.821</td>
<td>0.782</td>
</tr>
<tr>
<td>6.3</td>
<td>2006-09-30</td>
<td>0.606</td>
<td>0.677</td>
<td>0.844</td>
<td>0.756</td>
<td>0.778</td>
<td>0.892</td>
<td>0.836</td>
</tr>
<tr>
<td>7.1</td>
<td>2009-02-21</td>
<td>0.618</td>
<td>0.743</td>
<td>0.870</td>
<td>0.809</td>
<td>0.804</td>
<td>0.896</td>
<td>0.853</td>
</tr>
<tr>
<td>8.10</td>
<td>2011-02-04</td>
<td>0.776</td>
<td>0.827</td>
<td>0.906</td>
<td>0.868</td>
<td>0.864</td>
<td>0.926</td>
<td>0.896</td>
</tr>
<tr>
<td>8.26</td>
<td>2016-11-30</td>
<td>0.992</td>
<td>0.958</td>
<td>0.987</td>
<td>0.972</td>
<td>0.977</td>
<td>0.999</td>
<td>0.987</td>
</tr>
<tr>
<td>8.28</td>
<td>2017-09-01</td>
<td>0.999</td>
<td>0.995</td>
<td>0.999</td>
<td>0.997</td>
<td>0.996</td>
<td>0.999</td>
<td>0.997</td>
</tr>
</tbody>
</table>

- Our solution αDiff is better than BinDiff
- αDiff with only the *intra-function feature* (αDiff-1f) is good enough
- In most cases, αDiff could find the best match in the top 1 result (@1)
- Closer versions have higher similarity scores.
please refer to our ASE’18 paper for other evaluation results
Summary

- BCSD is fundamental to many applications.

- Traditional solutions have severe limitations.

- AI-based solutions could extract features of binary code, and improve the accuracy of BCSD.
Part 2: Security Threats to AI:

- New tech/application
- New threats
Deep Learning Systems Are Easily Fooled

Question: adversarial examples

- **White-box Attack**
  - Fast Gradient Sign (FGS)
  - Basic Iterative Method (BIM)
  - Jacobian-based Saliency Map (JSMA)
  - Carlini & Wagner Attack (CW)
  - DeepFool

- **Black-box Attack**
  - Particle Swarm Optimization (PSO)
  - Differential Evolution (DE)
  - ZOO
  - Decision-based attack

- **Common idea**
  - Optimization problem
  - Gradient descent

- **Better scalability**
  - Poor performance
Targeted Adversarial Attack
How to find TARGETED attack examples?

- The question
  - I of class Source $\rightarrow$ I’ of class Target

- This question is like **finding a vulnerability** in the neural network.

![Finding vulnerabilities in programs](image1)

![Finding vulnerabilities in neural network](image2)
Find vulnerabilities in programs

- Coverage-guided fuzzing (CGF)

- Adds testcases with new coverage into the seed pool
- Increases the coverage gradually during fuzzing
Find vulnerabilities in **Neural Networks**

- Coverage-guided fuzzing (CGF)
Find vulnerabilities in **Neural Networks**

- Coverage-guided fuzzing (CGF)
Power Scheduling

Prioritize seeds to mutate, and allocate energy for mutation.
Power Scheduling

Policy #1: Seeds closer to TARGET class are assigned with higher probability.
Power Scheduling

Policy #2: Seeds newly added to the pool are assigned with higher probability.

I am new here, my probability should be higher!
Power Scheduling

Policy #3: Seeds *seldomly chosen* are assigned with higher probability.

Hey, I need more attention!
Power Scheduling

- Policy #4: random choosing.

Fine, why not randomly select?
Find vulnerabilities in **Neural Networks**

- Coverage-guided fuzzing (CGF)
Mutation (Add Noise)
Mutation (Blur)
Mutation (Transformation)

... and many other mutation choices
Find vulnerabilities in **Neural Networks**

- Coverage-guided fuzzing (CGF)
Targeted Adversarial Example Generation

- Target: [0,0,0,0,1,0,0,0,0,0]
Targeted Attack Metrics

- Target: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

We want to get closer to the target

Target: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

Euclidean distance

[0.1, 0.05, 0.05, 0.03, 0.5, 0.04, 0.08, 0.12, 0.02, 0.01]
Targeted Attack Metrics

- Target: \([0,0,0,0,1,0,0,0,0,0]\)

  Euclidean distance

  \([0.1, 0.05, 0.05, 0.03, 0.5, 0.04, 0.08, 0.12, 0.02, 0.01]\)

  We want to get closer to the target
Targeted Attack Metrics

- Target: $[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]$  

**We want to get closer to the target**

- Focus on the prime component

Closer seeds will be prioritized during fuzzing, making it easier finding TARGETED adversarial examples.
Find vulnerabilities in **Neural Networks**

- Coverage-guided fuzzing (CGF)
Coverage Tracking

- Neuron coverage
- k-multisection Neuron Coverage,
- Neuron Boundary Coverage,
- ...

How many neurons have been activated?
Preliminary Result

Datasets: MNIST, CIFAR-10.

MNSIT (1→0)  CIFAR-10 (car→plane)
Evaluation

S: Success Rate
T: Time
R: Robustness
P: Perturbation

MNIST

CIFAR-10

FGSM
BIM
JSMA
CW
Our Method (CGF)
Evaluation: Success Rate of TARGETED Attacks

- Dataset: 500 instances on test set of MNIST (left) and CIFAR-10 (right)
- Time Threshold: 15s
- Epochs threshold: 10000

CW and Our method have a high success rate on MNIST!
Evaluation

MNIST

CIFAR-10

S: Success Rate
T: Time
R: Robustness
P: Perturbation

FGSM
BIM
JSMA
CW
Our Method (CGF)
Evaluation: Time Spent to Find First Adv Example

- Dataset: 500 instances on test set of MNIST and CIFAR-10
- Time Threshold: 15s
- Epochs threshold: 10000

CW and Our method are efficient on CIFAR-10!
Evaluation

MNIST

CIFAR-10

S: Success Rate
T: Time
R: Robustness
P: Perturbation

FGSM
BIM
JSMA
CW
Our Method (CGF)
**Evaluation: Robustness (resilient to pixel loss)**

- Dataset: 500 instances on test set of MNIST and CIFAR-10
- Time Threshold: 15s
- Epochs threshold: 10000

Our method and JSMA are more robust than others!
Evaluation

S: Success Rate
T: Time
R: Robustness
P: Perturbation

MNIST

CIFAR-10

FGSM
BIM
JSMA
CW
Our Method (CGF)
Evaluation: Perturbation (noise added)

- Dataset: 500 instances on test set of MNIST and CIFAR-10
- Time Threshold: 5s
- Epochs threshold: 10000

The Perturbation (L2)

\[ L_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \]

JSMA and CW are relatively better!
Evaluation

MNIST

CIFAR-10

S: Success Rate
T: Time
R: Robustness
P: Perturbation
Summary

- Adversarial example is a severe threat to machine learning.

- In addition to optimization-based solutions, coverage-guided fuzzing is also useful in generating TARGETED adversarial examples.
Conclusion

- AI is useful for security analysis.

- AI is facing severe security threats, and traditional security analysis solutions could help analyze AI models as well.

- More topics are to be explored in this field.
Thanks!

Q&A