Ph.D. Oral Defense

Software Obfuscation with Layered Security

Ph.D. Candidate: Hui Xu
Thesis Supervisor: Michael R. Lyu
Thesis Committee: Qiang Xu,
Patrick Lee,
Jiannong Cao

Sep 21st, 2018
The Problem of Software IP Protection

- Software intellectual property:
  - Server side (secure) ✓
  - Client side (vulnerable) ✗

- MATE (Man-At-The-End) attack [Collberg’11]: reverse engineer

- Examples of MATE attacks:
  - Disable License Checking
  - Clone Codes
  - Steal Algorithms

if (verifyLicense (key))
    startProgram();
else{
    printf ("invalid key");
    exit(-1);
}

Software Obfuscation for IP Protection

- Transform codes to a new version:
  - Difficult to read.
  - Preserve the semantics.
  - Incur little overhead.

Examples of control-flow obfuscation [Yadegari’15]

Example of lexical obfuscation

Critical Challenges for Obfuscation

- **Problem:** Obfuscation is not as secure as other security primitives.

- **Questions:**
  - What is the best security capability of software obfuscation?
  - How to design reliable obfuscation solutions?
Survey Results: Theoretical Area

❑ [Barak’01]: **Negative result** for black-box obfuscation.

❑ [Garg’13]: **Positive result** for indistinguishability obfuscation based on **graded encoding** (noisy multi-linear maps).

❑ Problems of graded encoding:
  - Only applicable to circuits: pure arithmetic.
  - Inefficient: polynomial overhead (several Gigabytes to obfuscate a 16-bit point function [Apon’14]).

Survey Results: Practical Area

- Most papers assume software written in particular languages, e.g., Java/C/Assembly.

- But real-world software is more complicated with heterogeneous components.

- Two types of software applications:
  - Client-server mode (e.g., Android applications).
  - Browser-server mode, i.e., web applications.
Example of Android Apps

RSA SecureID Software Token

Components
Example of Web Applications

Performance RNN
Why Hard for Protection?

[Bichsel’16]: We can recover a large portion of lexical information based on the residual information, e.g., names of invoked methods and strings.

Layered Security

- Principle: Swiss cheese model:
  - Mitigate risks through different layers.
  - Avoid single point of failure.

- Employed in aviation safety, healthcare, etc.
  - Safety-critical or security-critical.
  - The risks cannot be fully avoided.

- Introduced in IATF3.1.

---

Each area has multiple layers of protections

[IATF’02] Information Assurance Technical Framework Release 3.1, Department of Defense, 2002
Layered Security for Software Obfuscation

Why?

- Software is very complicated.
- Secure-against-all obfuscation techniques do not exist.

How?

- Based on risk management.
- Integrate multiple obfuscation techniques to mitigate risks.
- Each obfuscation technique only corresponds to particular threats.
Thesis Contributions

❑ **Develop a taxonomy** of obfuscation for layered security.
  ▪ Assist developers in designing layered obfuscation solutions.

❑ **Enrich the taxonomy** with three novel obfuscation techniques.
  ▪ *Symbolic Opaque Predicates*
    • Enhance the security of control-flow obfuscation.
  ▪ *N-version Obfuscation*
    • Enable the software with resilience to large-scale tampering attacks.
  ▪ *Deep Learning Obfuscation*
    • Application-level obfuscation technique for deep learning software.
Taxonomy for Layered Obfuscation

**What to obfuscate?**
- Code-Element Layer
  - Obfuscating Layout
  - Obfuscating Data
  - Obfuscating Controls
  - Obfuscating Methods
- Cross-Component Layer
- Application Layer
  - Obfuscating DRM Systems
  - Obfuscating DNN

**How to obfuscate?**
- Meaningless Identifiers
- Stripping Redundant Symbols
- Separation of Related Codes
- Junk Codes
- Symbolic Opaque Predicates
- Bogus Control Flows
- Probabilistic Control Flows
- Dispatcher-based Controls
- Implicit Controls
- White-box Encryption
- Knowledge Distillation
- Deep Learning Obfuscation

N-version Obfuscation

Application

Component Elements

Component Elements
Symbolic Opaque Predicates

- Obfuscating Layout
- Obfuscating Data
- Obfuscating Controls
- Meaningless Identifiers
- Stripping Redundant Symbols
- Separation of Related Codes
- Junk Codes
- Bogus Control Flows
- Symbolic Opaque Predicates
- Probabilistic Control Flows
- Dispatcher-based Controls
- Implicit Controls
- White-box Encryption
- Knowledge Distillation
Opaque Predicate

- **Definition** [Collberg’97] :
  - The value is known before compilation time.
  - Reverse analysis is difficult.

- **Application**: Control-flow obfuscation.

Vulnerable Example 1

Problem: Real-world opaque predicates are vulnerable.

```c
int a, b;  //b>0
...
func() {
...      
  if (a > b) {
    if ((a*(a+1))%2 == 0)
      fp = A[((fp)%2)+2];
    else
      fp = A[((fp)%2)+4];
  }
  else
    fp = A[((fp)%2)+3];
}
```

Example in [Ogiso’03]

Vulnerable to automated program analysis tools.

Vulnerable Example 2

Default opaque predicate generated by Obfuscator-LLVM [Junod’15]

**LLVM IR Code:**

```llvm
@x7 = common global i32 0
@y8 = common global i32 0
... 
define i32 @main() #0 {
  %1 = load i32* @x7
  %2 = load i32* @y8
  %3 = sub i32 %1, 1
  %4 = mul i32 %1, %3
  %5 = urem i32 %4, 2
  %6 = icmp eq i32 %5, 0
  %7 = icmp slt i32 %2, 10
  %8 = or i1 %6, %7
  br i1 %8, label %originalBB, label %originalBBalteredBB
}
```

**Source Code:**

```c
int x7 = 0;
int y8 = 0;

if((x7*(x7 - 1))%2 == 0||y8<10) {
  originalBB;
} else {
  originalBBalteredBB;
}
```


Constantly True
Adversarial Symbolic Execution

- Symbolic execution can detect opaque predicates by traversing the control-flow graph.

```c
int toy_func(int i) {
    if (i > 100) {
        return -1;
    }
    if (i * (i+1)%2!=0) {
        Bogus();
    }
    return 0;
}
```

Source Code

Control-flow Graph
Symbolic Execution Steps

- **Round 1:**
  Random Input: 
  \( i = 1000 \)
  
  **Entry:**
  \( \text{if}(i > 100) \)
  
  \( \text{T} \rightarrow \text{return } -1; \)

- **Round 2:**
  Constraint: \( NOT \ (i > 100) \)
  \( \Rightarrow i = 0 \)
  
  **Entry:**
  \( \text{if}(i > 100) \)
  
  \( \text{T} \rightarrow \text{return } -1; \)
  \( \text{F} \rightarrow \text{Return } 0; \)

- **Round 3:**
  Constraint:\( NOT(i > 100) \)
  \( \text{AND } i*(i+1)%2!=0 \)
  \( \Rightarrow \text{No Solution} \)
  
  **Entry:**
  \( \text{if}(i > 100) \)
  
  \( \text{F} \rightarrow \text{if}(i*(i+1)%2!=0) \)
  
  \( \text{F} \rightarrow \text{Return } 0; \)

It implies opaque predicates and bogus codes.
Adversary Model

The constraints are in the form of CNF: $\psi_1 \land \psi_2 \land \ldots \land \psi_n$

Our Objective and Approach

- **Objective** of Symbolic Opaque Predicates:
  - Enhance the security of opaque predicates.
  - Combat symbolic execution-based attackers.

- **Approach:**
  - Step 1: Investigate the limitations of symbolic execution tools.
  - Step 2: Employ these limitations to obfuscate software.
<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic Variable Declaration</td>
<td>Contextual variables other than arguments</td>
</tr>
<tr>
<td>Covert Propagations</td>
<td>Propagating symbolic values in covert ways</td>
</tr>
<tr>
<td>Buffer Overflows</td>
<td>Without proper boundary check</td>
</tr>
<tr>
<td>Parallel Executions</td>
<td>Processing symbolic values in parallel codes</td>
</tr>
<tr>
<td>Symbolic Memories</td>
<td>Symbolic values as the offset of memories</td>
</tr>
<tr>
<td>Contextual Symbolic Values</td>
<td>Retrieving contextual values with symbolic values</td>
</tr>
<tr>
<td>Symbolic Jumps</td>
<td>Symbolic values as the address of jump</td>
</tr>
<tr>
<td>Floating-Point Numbers</td>
<td>Symbolic values in float/double</td>
</tr>
<tr>
<td>Arithmetic Overflows</td>
<td>Beyond the scope of an integer type</td>
</tr>
<tr>
<td>Loops</td>
<td>Change symbolic values within loops</td>
</tr>
<tr>
<td>Crypto Functions</td>
<td>Processing symbolic values with crypto functions</td>
</tr>
<tr>
<td>External Function Calls</td>
<td>Processing symbolic values with external functions</td>
</tr>
</tbody>
</table>

Example of Symbolic Memories

- Use symbolic values as the offsets to access memory.
- Theoretical challenge: some **pointer analysis** problems are NP-hard.

```c
void func(int symvar)
{
    int l1_ary[] = {1,2,3,4,5,6,7};
    int l2_ary[] = {symvar,1,2,3,4,5,6,7};
    int i = l2_ary[l1_ary[symvar%7]];
    if(i == 1)
        foobar();
}
```

*Can symbolic execution tools find a test case for triggering foobar()?*
Example of Floating-point Numbers

- Floating-point numbers are approximations of real numbers.
- Defined in IEEE-754: Interval for 32-bit float: \([1.401298464324817e-45, 32767.9990234]\.\)

```c
void func(int symvar) {
    float f = symvar / 1000000.0;
    if (f == 0.1) {
        bogus();
    }
    if (1024 + f == 1024 && f > 0) {
        foobar();
    }
}
```
Examining the Prevalence of Challenges

hxuhack / logic_bombs

- buffer_overflow
- contextual_symbolic_value
- covert_propogation
- crypto_functions
- external_functions
- floating_point
- integer_overflow
- loop
- parallel_program
- symbolic_jump
- symbolic_memory


Developed at UCSB (2016) http://angr.io/

Benchmarking Results

These tools failed most of the test cases.

<table>
<thead>
<tr>
<th>Challenge</th>
<th>KLEE</th>
<th>Triton</th>
<th>Angr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic Variable Declaration</td>
<td>0/7</td>
<td>0/7</td>
<td>0/7</td>
</tr>
<tr>
<td>Covert Propagations</td>
<td>1/5</td>
<td>1/9</td>
<td>4/9</td>
</tr>
<tr>
<td>Buffer Overflows</td>
<td>0/4</td>
<td>0/4</td>
<td>2/4</td>
</tr>
<tr>
<td>Parallel Executions</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
</tr>
<tr>
<td>Symbolic Memories</td>
<td>5/6</td>
<td>0/8</td>
<td>7/8</td>
</tr>
<tr>
<td>Contextual Symbolic Values</td>
<td>0/7</td>
<td>0/7</td>
<td>0/7</td>
</tr>
<tr>
<td>Symbolic Jumps</td>
<td>1/1</td>
<td>0/4</td>
<td>2/4</td>
</tr>
<tr>
<td>Floating-Point Numbers</td>
<td>0/5</td>
<td>0/5</td>
<td>2/5</td>
</tr>
<tr>
<td>Arithmetic Overflows</td>
<td>2/2</td>
<td>1/2</td>
<td>2/2</td>
</tr>
<tr>
<td>Loops</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
</tr>
<tr>
<td>Crypto Functions</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
</tr>
<tr>
<td>External Function Calls</td>
<td>0/8</td>
<td>1/8</td>
<td>3/8</td>
</tr>
<tr>
<td>Total</td>
<td>9/54</td>
<td>3/63</td>
<td>22/63</td>
</tr>
</tbody>
</table>
Design Symbolic Opaque Predicates

Opaque Predicate Engine

Choose a Parameter

Design a Challenging Problem

Create Opaque Predicates

Code Snippet (Function) ➔ input

Template Repository ➔ output

Opaque Predicates
Template of Symbolic Memories

1 int func(int i) {
2     if (i == 7) {
3         foobar();
4     }
5 }

1 int func(int i) {
2     int l1_ary[] = {1,2,3,4,5,6,7};
3     int l2_ary[] = {j,1,2,3,4,5,6,7};
4     int j = l2_ary[l1_ary[i%7]];
5     if (i == j) {
6         if (j == 1 && i == 7) {
7             foobar();
8         } else {
9             bogus();
10         }
11     }
12 }

Bi-opaque

Obfuscated Code

Type I opaque predicate

Type II
Evaluation

- **Performance Metrics:**
  - Space overhead
  - Execution overhead

- **Target Programs:**
  - Linux Busybox (e.g., cat)
  - Encryption programs (e.g., AES)

- **Prototype** based on Obfuscator-LLVM.
Space Overhead

Obfuscating Linux-cat

Baseline: Obfuscator-LLVM

Efficient with dynamic linkage

# of replaced opaque predicates

Program Size (KB)

Symbolic Memory
Floating-point Number
Covert Propagation
Parallel Programming

0 1 5 10 all

Motivation
Approach
Evaluation
Execution Overhead

Obfuscating Linux-cat

Be careful of inefficient codes

Obfuscating AES

Amplified in loops
Summary of Symbolic Opaque Predicates

Objective:

- To secure control-flow obfuscation against symbolic execution.

Our Contribution:

- We investigated the vulnerabilities of symbolic execution and developed a dataset to benchmark symbolic execution tools.
- We proposed a framework to compose opaque predicates leveraging these vulnerabilities.

Current Work:

- Enrich the template repository with more diversified samples.
- Develop a systematic strategy of opaque predicate insertion with small overhead.
N-version Obfuscation

Software Obfuscation

Code-Element Layer
- Obfuscating Layout
- Meaningless Identifiers
- Stripping Redundant Symbols
- Separation of Related Codes
- Junk Codes

Software-Component Layer
- Obfuscating Data
- Obfuscating Controls

Cross-Component Layer
- Obfuscating Methods
- Bogus Control Flows
- Probabilistic Control Flows
- Dispatcher-based Controls
- Implicit Controls

Application Layer
- Obfuscating DRM Systems
- White-box Encryption
- Knowledge Distillation
- Obfuscating DNN
Motivation

- Software tampering attack is popular for smartphones, especially Android.

**Static App Repack:**

- **unpack**
- **apktool**
- **apk file**
- **repack**
- **apktool**
- **tampered apk**

**Smali code**

**Dynamic Injection:** 7/10 Android security apps in China inject payloads into their “protected” apps.

---

We can have a bunch of solutions, but none is overwhelming.

“Given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application.”

--Christian Collberg
Our Objective and Approach

- **Objective** of N-version obfuscation:
  - Defend apps against software tampering attack.
  - We focus on **impeding large-scale attacks only**.

- **Approach**:
  - Create diversified apps for different clients.
  - Impede the replication of an attack on multiple hosts.
A Candidate Solution for Networked Apps

**Motivation**

**Approach**

- **Diversified components**
- **Diversified algorithms**

**App Client 1**
- Safeguard
- Integrity Checking
- Message Authentication

**App Client 2**
- Safeguard

**App Client N**
- Safeguard

**Server**
- MAC
- Hash Parent
- N-version Database

- request → MAC(request) → response
- DAI → param
Diversified MAC Algorithms based on SHA1

Data: $w[80]$

\[\begin{align*}
// & \text{ blocks of plaintext} \\
\text{for } i = 0; i < 80; i++ & \text{ do} \\
& \text{if } 0 \leq i \leq 19 \text{ then} \\
& \quad f \leftarrow (b \text{ AND } c) \text{ OR } ((\text{NOT } b) \text{ AND } d); \\
& \quad k \leftarrow 0X5A827999; \\
& \text{end} \\
& \text{if } 20 \leq i \leq 39 \text{ then} \\
& \quad f \leftarrow b \text{ XOR } c \text{ XOR } d; \\
& \quad k \leftarrow 0X6ED9EBA1; \\
& \text{end} \\
& \text{if } 40 \leq i \leq 59 \text{ then} \\
& \quad f \leftarrow (b \text{ AND } c) \text{ OR } (b \text{ AND } d) \text{ OR } (c \text{ AND } d); \\
& \quad k \leftarrow 0X8F1BBCDC; \\
& \text{end} \\
& \text{if } 60 \leq i \leq 79 \text{ then} \\
& \quad f \leftarrow b \text{ XOR } c \text{ XOR } d; \\
& \quad k \leftarrow 0XCA62C1D6; \\
& \text{end} \\
\text{temp} \leftarrow (a \text{ LEFTROTATE } 5) + f + e + k + w[i]; \\
e \leftarrow d; \\
d \leftarrow c; \\
c \leftarrow b \text{ LEFTROTATE } 30; \\
b \leftarrow a; \\
a \leftarrow \text{temp};
\end{align*}\]

Each client employs a random combination

Motivation

Approach

Data: $f\_genes[80], k\_genes[80], w[80]$

\[\begin{align*}
\text{for } i = 0; i < 80; i++ & \text{ do} \\
& \text{Call } f\_genes[i]; \\
& // \text{ Pointer to } F0, F1, F2 \text{ or } F3 \\
& \text{F\_TAIL}(k\_genes[i], w[i]); \\
& \text{end} \\
\text{Function } F0() \\
& \quad f \leftarrow (b \text{ AND } c) \text{ OR } ((\text{NOT } b) \text{ AND } d); \\
\text{Function } F1() \\
& \quad f \leftarrow b \text{ XOR } c \text{ XOR } d; \\
\text{Function } F2() \\
& \quad f \leftarrow (b \text{ AND } c) \text{ OR } (b \text{ AND } d) \text{ OR } (c \text{ AND } d); \\
\text{Function } F3() \\
& \quad f \leftarrow b \text{ XOR } c \text{ XOR } d; \\
\text{Function } F\_TAIL(k, w) \\
& \quad \text{temp} \leftarrow (a \text{ LEFTROTATE } 5) + f + e + k + w; \\
& \quad e \leftarrow d; \\
& \quad d \leftarrow c; \\
& \quad c \leftarrow b \text{ LEFTROTATE } 30; \\
& \quad b \leftarrow a; \\
& \quad a \leftarrow \text{temp};
\end{align*}\]
Feasibility of Automation

- Automation of N-version Generation:
  - Can be implemented as a compiler pass.

- Automation of N-version Delivery:
  - Server delivers the safeguard as a dynamic library to each client at the first time of launch.
  - Clients register their versions on the server.
Summary of N-version Obfuscation

**Objective:**
- To defend software against tampering attacks.
- We focus on impeding large-scale attacks.

**Our Contribution:**
- We proposed an N-version obfuscation solution for networked apps.
- It is efficient to automatically generate and deliver N software versions.
Our Proposed Approaches in the Taxonomy

Software Obfuscation

- Code-Element Layer
  - Obfuscating Layout
    - Meaningless Identifiers
    - Stripping Redundant Symbols
    - Separation of Related Codes
    - Junk Codes
  - Obfuscating Data
  - Obfuscating Controls
    - Bogus Control Flows
    - Probabilistic Control Flows
    - Dispatcher-based Controls
    - Implicit Controls
  - Obfuscating Methods
  - Cross-Component Layer
    - ...
Motivation

- Running deep learning models on client sides is a trend.

- Deep learning models are vulnerable to piracy.

https://datascience.stackexchange.com/questions/13175/neural-network-obfuscation
Structure Piracy

- Structure is the key factor for improving accuracy.

Top-5 Error (%) in ILSVRC

- AlexNet 2012 winner
- VGG 2014 runner up
- GoogLeNet 2014 winner
- ResNet 2015 winner

GoogLeNet Inception Module

ResNet Inception Module

DenseNet Inception Module (2017)
Parameter Piracy

- Employ a well-trained model as the initial state to create new models.
  - Transfer learning.
  - Incremental learning.

### Motivation

- **Transfer Learning Experiment**
  - CIFAR-10 to STL10
  - ResNet (58.67%)
  - GoogLeNet (58.49%)

- **Incremental Learning Experiment**
  - CIFAR-10 to CIFAR-100
  - Incremental Learning (67.95%)
Our Objective and Approach

- **Objective**: defend deep learning models against piracy attacks.
  - Structure piracy.
  - Parameter piracy.

- **Approach**: simulate the model with a shallow network.
  - Combat structure piracy by hiding the critical structures.
  - Combat parameter piracy by degrading the learning ability.
  - We should obtain a simulation network with zero accuracy loss.
    - Recursive simulation.
    - Joint training.
Recursive Simulation of GoogLeNet

**Motivation**

**Approach**

- **GoogLeNet**
  - 19 conv layers

- **Intermediate Result**
  - 9 conv layers

- **Obfuscated GoogLeNet**
  - 7 conv layers
Features should be computed from the same (or super) set of pixels.

Kernel size of the simulation network:

Compress 2 layers to 1 layer: \( h = h_1 + (h_2 - 1) \times s_1 \)

Compress \( n \) layers to 1 layer: \( h = h_1 + (h_n - 1) \times \cdots \times (h_2 - 1) \times s_{n-1} \times \cdots \times s_1 \)
Joint Training

- An improvement based on the teacher-student network.
  - The loss of student network cannot be zero.
  - The teacher network itself has errors.
Overall Framework of Obfuscation

Training Phase

First Simulation Round
- Define an intermediate network
- Train
- New param.
- Merge
- Obfuscate each block iteratively

Second Simulation Round
- Define an intermediate network
- Train
- Intermediate result
- Fine-tune
- Input
- Merge
- Obfuscated model

Motivation

Approach
Evaluation Experiments

- **Evaluation Purposes:**
  - Accuracy: can we obfuscate the model with zero accuracy loss?
  - Overhead: size and execution cost.
  - Security: resilience to parameter piracy.

- **Models for Obfuscation:**
  - GoogLeNet, ResNet, and DenseNet trained with CIFAR-10.
  - ResNet and DenseNet trained with ImageNet (five classes).
# Results of Obfuscated Models

<table>
<thead>
<tr>
<th>Model (CIFAR-10)</th>
<th>Performance</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Size (MB)</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>90.83%</td>
<td>2.51</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>90.92%</td>
<td>2.49</td>
</tr>
<tr>
<td>ResNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>90.94%</td>
<td>43.36</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>91.04%</td>
<td>11.38</td>
</tr>
<tr>
<td>DenseNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>90.14%</td>
<td>4.24</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>90.31%</td>
<td>4.21</td>
</tr>
</tbody>
</table>

- No accuracy loss.
- More efficient.

<table>
<thead>
<tr>
<th>Model (ImageNet)</th>
<th>Performance</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Size (MB)</td>
</tr>
<tr>
<td>ResNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>92.4%</td>
<td>43.37</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>92.4%</td>
<td>36.72</td>
</tr>
<tr>
<td>DenseNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>91.6%</td>
<td>4.27</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>92.8%</td>
<td>2.94</td>
</tr>
</tbody>
</table>
Resilience to Parameter Piracy

- The accuracy of pirated models based on the obfuscated models declines obviously than based on the original ones.

<table>
<thead>
<tr>
<th>Model</th>
<th>Incremental Learning (CIFAR-10 to CIFAR-100)</th>
<th>Transfer Learning (CIFAR-10 to STL10)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Degradation</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>66.5%</td>
<td>-</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>63.59%</td>
<td>-4.4%</td>
</tr>
<tr>
<td>ResNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>66.92%</td>
<td>-</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>64.77%</td>
<td>-3.2%</td>
</tr>
<tr>
<td>DenseNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>67.16%</td>
<td>-</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>62.91%</td>
<td>-6.3%</td>
</tr>
</tbody>
</table>
Objective: to secure deep learning models against piracy.

Our Contribution:

- We proposed a simulation-based obfuscation approach.
- We conducted real-world experiments and achieved promising results.
  - No accuracy loss.
  - No overhead.
  - Resilient to parameter piracy.
Conclusion

❑ We proposed layered obfuscation as a promising way for software obfuscation.

❑ We presented a taxonomy of obfuscation techniques for layered obfuscation.

❑ We discussed three novel obfuscation techniques.
  ▪ Symbolic opaque predicates.
  ▪ N-version obfuscation.
  ▪ Deep learning obfuscation.
Future Work

- Practice layered obfuscation with more real-world software.
- Develop a methodology for implementing layered obfuscation.
- Propose new obfuscation techniques for new security issues.
- Develop a practical obfuscation tool integrating multiple techniques.
Publications Related to the Thesis

❑ Motivation and Taxonomy (Chapter 2):


❑ Newly Proposed Approaches (Chapter 3, 4, 5):


Other Publications Related to Software Engineering and Cybersecurity


