Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.2, Apr. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.2.233

자연어 처리를 통한 코드 난독화 커버리지 측정

김 병 연,[1†] 김 휘 강[2‡]
[1,2]고려대학교 정보보호대학원 (대학원생, 교수)

Quantitative Measures for Code Obfuscation Coverage by the Natural Language Processing

Byeong Yeon Kim,[1†] Huy Kang Kim[2‡]
[1,2]Korea University School of Cybersecurity (Graduate student, Professor)

요 약

난독화는 코드를 보호하고 분석을 위해 더 큰 노력을 요구하기 위한 목적으로 일반 앱부터 악성 앱까지 광범위하게 사용되고 있다. 따라서 공격자와 보안 담당자는 보안성 분석을 위해 앱이 어느 정도 난독화 되어있는지 아는 것이 중요한데, 현재 관련 연구 및 솔루션들의 성능은 좋지 않다. 첫 번째로 상용 솔루션들은 조금의 난독화만 발견해도 전체가 난독화 되었다고 판단하고 있다. 두 번째로, 읽을 수 있지만 이해할 수 없는 방식의 난독화를 발견하지 못한다. 마지막으로, 자체적으로 비공개 난독화 기술을 개발하여 난독화 하는 기업들도 생겨나고 있으므로 단순히 시중에 존재하는 난독화 도구의 규칙을 학습하는 기존 방법으로는 난독화를 탐지하는 것에 한계가 있다. 따라서 본 논문에서는 소스 코드를 문서처럼 학습하여 '코드를 얼마나 읽을 수 있는지'에 대한 것을 넘어서서 '얼마나 이해할 수 있는지'에 대한 관점으로 접근하였고, 자연어 처리, 휴리스틱을 통해 코드 난독화 구역을 측정할 수 있는 솔루션 "AndrObfusec"를 개발하여 높은 정확도로 난독화를 분류해 냈다.

ABSTRACT

Obfuscation has been vastly applied to both malware and benign Android applications in the last years. Because Obfuscation hides the apps' semantics from analysts by increasing the cost of reverse engineering and decompilation. Consequently, It is important for attackers and security team to measure the quantitative of obfuscation of the app for analysis. However, current research and solutions are surprisingly bad at detecting obfuscation.

First, When only a small amount of obfuscation is found, They will have the tendency to judge that code as obfuscated. Second, They can not detect misunderstandable obfuscation techniques.

Finally, The systems do not necessarily remain effective over time – when novel obfuscation techniques are proposed.

In this work, we propose AndrObfusec, an Natural language processing and heuristic based system to detect obfuscation in Android applications, known as identifier renaming. This system examines a different aspect of the issue - It measure not only readability but also understandability with quantitative measurement for code obfuscation coverage. Particularly, AndrObfusec achieves an high accuracy for identifier renaming detection.

Keywords: Code Obfuscation, Obfuscation Coverage, Natural Language Processing

Received(01. 28. 2021), Modified(03. 11. 2021), Accepted(03. 12. 2021)

† 주저자, securekim@korea.ac.kr
‡ 교신저자, cenda@korea.ac.kr(Corresponding author)

I. 서 론

안드로이드 시스템의 경우 쉽게 디컴파일해서 소스 코드를 볼 수 있으므로, 난독화 및 탐지가 중요해졌다. 따라서 기존 연구들은 난독화를 탐지하기 위해 어떤 난독화 툴로 난독화 되어있는지 식별하였다. 하지만 지금, 이 순간에도 계속 새로운 난독화 툴이 개발되고 있으며, 규모가 큰 기업들의 경우 자체 개발한 비공개 난독화 툴을 사용하는 때도 있으므로 기존 방식으로 난독화를 탐지하는 데는 한계가 있다. 또한, 기존 연구들은 단순히 어떤 난독화 툴로 난독화 되었다는 사실만을 식별하는 데 그쳤는데, 보안 담당자와 공격자에게 중요한 것은 어떤 툴로 난독화 되어 있는지가 아닌 얼마나 난독화 되어있는 지이다.

보안 담당자 관점에서 얼마나 난독화 되어있는지 필요한 구체적인 이유는 다음과 같다. 국내의 경우 IT 인력 중 정보보호를 담당하는 인력을 보유한 사업체가 약 20%밖에 되지 않는, 개발자보다 보안 전문가가 현저히 적은 상황이기 때문에 개발자가 어느 정도 보안 점검을 진행할 수밖에 없다[2]. 그런데 개발자가 수동으로 검사하기에는 전문성 및 생산성이 떨어지고, 잦은 빌드와 마켓 배포의 격차로 인해 제대로 난독화가 되지 않은 앱을 배포하는 때도 있고 툴의 설정마다 난독화 Coverage가 달라지기 때문에 마켓에 배포하기 전에 난독화가 얼마나 되어있는지 확인하는 방법이 필요하다. 특히 개발에 관여하지 않고 보안성 평가만 하는 경우 기존 코드를 알 수 없는 블랙박스로 진행되기 때문에 난독화가 되어있더라도 기존 코드보다 얼마나 난독화 되어있는지 쉽게 알기 어렵다.

반면 공격자로서도 앱이 얼마나 난독화 되어있는지 아는 것이 중요하다. 난독화가 잘 되어있는 앱은 ROI(Return Of Investment)가 낮아지기 때문에 공격 우선순위를 낮추게 되며, 반대로 난독화가 되어있더라도 Coverage가 낮은 앱이라면 공격을 시도해 볼 수 있으므로 공격자 또한 앱의 난독화도를 체크 할 방법이 필요하다. 따라서 본 논문에서는 코드를 이해하는 관점에서 접근하여 각 Package, Class, Method, Variable을 전처리하고 학습하여 난독화도를 정량적으로 측정할 방안을 제시하고, 상용앱에 적용하여 측정해 본다.

II. 관련 연구

2.1 배경 지식

Parvez Faruki et al[3]은 안드로이드 난독화에 대해 수십 편의 논문과 방법을 다방면으로 조사하였으며, 이를 요약하면 크게 세 가지 방법으로 나눌 수 있다.

2.1.1 Logic Obfuscation

(1) Control Flow Computation : 실행 시 닿지 않는 죽은 코드를 넣거나, 조건을 추가하거나, 연산을 추가하거나, 프로세스를 나누어 난독화 하는 방법
(2) Control Flow Aggregation : 코드를 인 라이닝 하거나, 함수와 파라미터를 병합하거나, 함수를 복사하거나, 루프를 변형하여 난독화 하는 방법
(3) Control Flow Ordering : 상태의 순서를 변경하거나, 루프를 반대로 구성하여 난독화 하는 방법
(4) Control Flow Flattening : 실행시간에 변경되는 새로운 분기를 추가해 분기의 대상을 판별하기 어렵게 하는 방법

2.1.2 Data Obfuscation

(1) Data aggregation : 둘 이상의 스칼라 변수를 하나로 병합하거나, 상속과 인터페이스를 사용해 클래스 참조를 복잡하게 하거나, 문자열을 분할 하거나 병합하여 읽기 힘들게 하는 방법
(2) Data Storage and Encoding : 로컬 변수를 전역 변수로 변환하거나 인코딩하거나 변수 표현 식을 대체하는 방법
(3) Data Ordering : 데이터 선언 순서를 무작위로 변경하는 방법

2.1.3 Layout Obfuscation

(1) Identifier Scrambling : 패키지, 클래스, 함수, 변수명 등을 랜덤한 알파벳으로 치환하

는 방법

(2) Identifier Renaming : 패키지, 클래스, 함수, 변수명 등을 의미 없는 식별자로 변경하여 식별하기 어렵게 만드는 방법

(3) Comments or Debug information : 임의의 주석, 디버그 정보를 추가하는 방법

이와 같이 난독화에는 크게 세가지 방법이 있는데, 본 논문에서는 퀄리티 측면이 아닌 정량적인 측면에 초점을 맞추어 레이아웃 난독화에 대해 자연어 처리를 통해 측정하는 방법을 다룬다. 따라서 아래에서는 Roedy green[1]이 작성한 내용을 바탕으로 잘못된 단어나 오타를 사용하여 이해하기 어려운 코드를 만드는 방법에 관해 좀 더 자세히 알아본다.

2.1.4 Identifier Obfuscation

(1) fred 등 태아 작명법으로 명명
(2) a 등 단일 문자 변수의 사용
(3) papor 등의 창의적인 오타 사용
(4) routine 등의 추상적인 단어 사용
(5) ComputeRasterHistoGram 등과 같이 의도적인 낙타 표기법의 변경
(6) 사용했던 이름의 재사용
(7) _ (Underbar) 의 무분별한 사용
(8) 인간의 언어와 컴퓨터 언어의 혼용
(9) ß 등 확장 아스키의 사용
(10) 영어가 아닌 다른 나라 언어의 혼용
(11) slash 등 기호를 나타내는 단어의 사용
(12) superman, starship 등 전혀 관계없는 이름의 사용
(13) l, 1, I 등 모양이 비슷한 소문자와 숫자, 대문자의 혼용
(14) blue 대신 lancelotsFavoriteColor 등 자신만 알아볼 수 있는 이름의 사용

2.2 난독화 탐지 관련 연구

위에서 설명한 난독화를 탐지하기 위한 연구는 활발하게 이루어지고 있다. 먼저 Mohsen 등은 어떤 문자열을 정의하는 표현 중 가장 짧은 표현을 의미하는 Kolmogorov 복잡성과 데이터 압축을 기반으로 기존 소스 코드와 난독화 한 결과를 이용해 43개의 난독화 기법에 대한 정량적인 측정을 시도했다[4]. 다만 이것은 난독화 하기 전의 개발 소스 코드가 있어야 측정이 가능한 방법이므로, 개발자가 아니거나 코드를 받을 수 없는 환경에서는 평가할 수 없다는 한계가 있다. 또한, 기계학습을 통해 어떤 툴로 난독화 되었는지 판단하고자 하는 연구도 있었다[5]. 이 연구에서는 Proguard 외 4가지 무료 난독화 툴을 KNN, DecisionTree, SVM, RandomForest, MLP, AdaBoost, GaussianNB, LogisticRegression 등으로 학습하고 어떤 툴로 난독화 되었는지 90% 정확도로 인식하였다. 다만 해당 연구는 앱 자체가 얼마나 난독화 되어있는지에 대한 부분을 밝혀내지 못했다. 반면 앱들이 얼마나 난독화가 되고 있는지 구글 플레이 마켓에 존재하는 앱들에 대해 광범위한 관점에서 진행한 연구도 있다[6]. 이들은 각종 난독화 도구에서 사용하는 룰을 기반으로 앱이 난독화 되었는지 판단하는 Obfuscan 이라는 툴을 만들고, 구글 마켓에 존재하는 약 180만 개의 앱들에 대해 약 25%만 난독화가 되고 있다는 사실을 밝혀낸 뒤 개발자에 대한 교육이 필요하다고 주장하였으나 마찬가지로 앱 자체가 얼마나 난독화 되어있는지는 밝혀내지 못했다. androODet는 기계학습을 이용해 식별자 이름 난독화에 대해 92%의 정확도로 난독화 여부를 판단한 연구이다[7]. 그러나 이 연구 또한 얼마나 난독화 되어있는지 판단하려는 목적은 아니며, 일반화 모델을 학습하는 것이 아닌 것으로 밝혀져 동일 제품군의 샘플이 데이터에 없는 경우 정확도가 50%까지 떨어진다는 사실이 다른 반박 연구를 통해 알려졌다[8]. 이러한 androODet의 문제를 개선했다고 주장하는 연구도 있었다[9]. 이 연구에서는 Naïve Bayes 방법을 기반으로 한 생성 접근 방식을 제안하였는데, 먼저 자연어 문자열 분포를 모델링 한 다음 언어를 문자열에 할당할 수 있는 신뢰도를 기준으로 분류함으로써 난독화 샘플을 사용하지 않고 난독화를 탐지할 수 있다고 주장하였다. 하지만 이 방법은 자연어로 되어있어 사람이 읽을 수는 있지만 이해되지 않는 방식의 난독화는 탐지할 수 없다는 한계가 있다.

따라서 본 논문에서는 기존에 해결되지 않은 '얼마나 난독화 되어 있는지'에 대한 문제를 '얼마나 이해할 수 없는지'의 관점으로 정량 측정한다.

2.2.1 상용 보안 솔루션

현재 상용화된 보안 솔루션에 난독화 탐지 기능이 얼마나 어떻게 구현되어있는지 확인하기 위해 Table 1.에 보안 솔루션들의 기능을 조사하여 정리하였다. 조사에 따르면 난독화를 탐지하는 기능은 솔루션 E와 솔루션 F에 있었다. 그런데 솔루션 E는 내부적으로 솔루션 F를 사용하므로, 결국 난독화를 탐지하기 위해 사용되는 것은 솔루션 F 하나였다.

F의 탐지 룰을 자세히 조사한 결과, Fig. 1.과 같이 단순히 패턴을 매칭시켜서 난독화가 어떤 툴로 되어있는지 찾는 방식이었고, 탐지 가능한 난독화 툴은 Dexguard, Dexprotector, Bitwise antiskid, Arxan, Allatori, Aamo, Appsuit, Gemalto, Kiwi에 불과했다.

따라서 본 논문에서 해결하고자 하는 난독화 방식과 얼마나 난독화 되어있는지 검사할 수 없었으며, 코드가 조금이라도 난독화 패턴과 맞는다면 앱 전체가 난독화 되어있다고 판단하게 된다는 한계점이 있었다.

```
rule arxan : obfuscator {
  meta:
    description = "Arxan"
    url         = "https://www.arxan.com/products/application-protection-mobile/"
    sample      = "7bd1139b5f860d48e0c35a3f117f980564f4\
    5c177a6ef480588b5b5c8165f47e"
    author      = "Eduardo Novella"
  strings:
    // Obfuscated Lpackage/class/: "L([a-z]\1{5}\/[a-z]{6}\/".
    // AFAIK, Yara does not support backreferences at the moment, thus this is the combo:
    $pkg = /L(a{6}|b{6}|c{6}|d{6}|e{6}|f{6}|g{6}|h{6}|i{6}|j{6}|k{6}|l{6}|m{6}|n{6}|o{6}|p{6}|q{6}|r{6}|s{6}|t{6}|u{6}|v{6}|w{6}|x{6}|y{6}|z{6})\/[a-z]{6}/
    // Obfuscated methods are found to follow a pattern like:
    // 1 byte size + 1 byte ASCII + [7-26] non-ASCII bytes + 00 ( null terminator)
    $m1 = { 10 62 (6? | 75) [14] 00 }
    $m2 = { (0b | 0d) 62 d0 [15] 00 }
    $m3 = { (0e | 10) 62 30 34 3? [15] 00 }
    $m4 = { (0b | 0d) 62 30 34 3? [13] 00 }
    $m5 = { (08 | 0b | 0d | 0e ) 62 [7-13] 00 }
    $m6 = { 0a 62 (30 34 3? | d? ?? ??) [11] 00 }
    $m7 = { (0d | 0b | 11) (62 d1 8? | 6? ?? ??) [14] 00 }
  condition:
    is_dex and
    $pkg and
    6 of ($m*)
    }
```

Fig. 1. Detection rule for Arxan obfuscator

Table 1. Mobile Security Solutions

	A	B	C	D	E	F
Enabled Application Backup	O	O		O		
Enabled Debug Mode	O	O	O	O		
External data in raw SQL queries	O		O			
Missing anti-emulation	O					O
Missing tapjacking protection	O	O				
Possible Man-In-The-Middle Attack	O	O				
Predictable Random Number Generator	O					
Usage of banned API functions	O	O				
Weak encryption	O	O		O		
Weak hashing algorithms	O	O		O		
Signer Certificate				O	O	
Shared Lib binary analysis				O		
Domain(URL) Malware check				O	O	
Emails				O	O	
Obfuscation					O	O

2.2.2 난독화 솔루션

이어서 난독화 시 어떤 방법이 사용되고 있는지 확인하기 위해 상용 소프트웨어의 목록과 기능을 Table 2.에 정리하였다.

그 결과, 난독화 솔루션들은 모두 이름을 단순히 읽을 수 없는 단어 형태로 치환하는 방식이었다. 그러나 앞서 소개한 것처럼 자연어를 의미가 있는 다른 자연어로 치환하여 난독화가 적용이 안 된 것으로 오판하게 하는 방식의 난독화 연구가 있었으며[10], 이론에 그치지 않고 Fig. 2.와 같이 의미가 없는 자연어로 치환하는 난독화 기법이 적용된 상용 프로그램도 실제로 존재했다.

위 코드는 통합 백엔드 서비스 플랫폼을 제공하는 Bmob이라는 회사의 라이브러리로, 내용을 보면 thing, Hamlet, yet, what, Code 등 추상적인 단어로 이루어져 사람이 읽을 수는 있지만, 동작과 내용에 아무런 연관성이 없어 전혀 이해할 수 없다.

Table 2. Features of Obfuscators

Obfuscator	Proguard	DashO	Dexguard	yGuard	DexProtector	Allatori
Name	O	O	O	O	O	O
String		O	O		O	O
Controlflow		O	O		O	O

```
1  package cn.bmob.push.lib.service;
2
3  import cn.bmob.push.lib.a.thing;
4  import cn.volley.Hamlet;
5  import cn.volley.yet;
6
7  final class what implements yet {
8      private /* synthetic */ This cj;
9
10     what(This thisR) {
11         this.cj = thisR;
12     }
13
14     public final void Code(Hamlet hamlet) {
15         new thing(this.cj.context).m();
16     }
17 }
```

Fig. 2. Example of obfuscated code

III. 제안 모델

3.1 아이디어

본 논문에서는 안드로이드 앱이 단순히 치환되는 방식으로 난독화 된 것뿐만 아니라, Fig. 2.에서 본 것처럼 사람이 읽을 수 없는 방식으로 난독화 되었다 하더라도 탐지하기 위해 코드를 문서처럼 생각해 보는 아이디어를 제시한다.

먼저 오픈소스는 개인이 아니라 여러 사람이 협업하기 위한 것이므로, 낙타 표기법 등 일반적인 Java Naming Convention[11]과 문법을 잘 지키리라 예측할 수 있다. 이렇게 오픈소스는 남을 쉽게 이해시키기 위한 문서이므로 자기소개서 문서 등과 비슷하다고도 생각해 볼 수 있다. 이러한 문서의 특성은 일반적으로 제목과 문단 제목이 연관성이 있고 문단과 그 문단에 들어가는 단어들이 연관성이 있다는 것이다. 이처럼 일반적인 코드도 패키지-클래스, 클래스-메소드, 메소드-변수가 연관성을 갖게 되므로, 단어 간 연관성이 전혀 없거나 너무 빈번하게 사용되면 해당 부분이 난독화 되어있다고 판단할 수 있을 것이다. 이러한 문서와 코드의 연관성을 나타낸 것은 Fig. 3.와 같다.

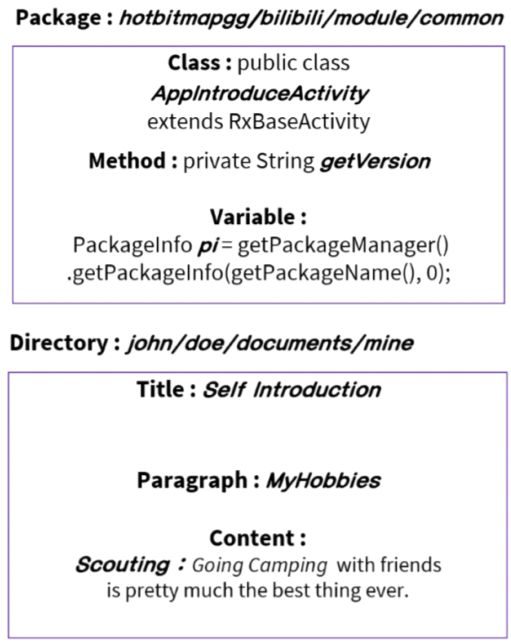


Fig. 3. Code VS Document

이러한 아이디어를 기반으로 제안 방법을 순서대로 설명한다. 먼저 오픈소스 코드를 수집하여 빌드한 APK 파일들을 디컴파일 한 뒤, Java 코드가 아닌 Smali 코드를 사용하여 패키지, 클래스, 메소드, 변수 단위로 추출하고 단어를 나누어 Vector 화한다.

다음으로 Smali 코드를 기준 선언 특징을 바탕으로 각 단어를 추출한다.

(1) Class 종류별 선언 특징
- .class
- .annotation system Ldalvik/annotation/MemberClasses;
- .annotation system Ldalvik/annotation/InnerClass;
- .annotation system Ldalvik/annotation/EnclosingClass;

(2) Method 선언 특징
- .method,
- method = {
- .annotation system Ldalvik/annotation/EnclosingMethod

(3) Variable 선언 특징
- .field (함수 외부 변수)
- .local (함수 내부 변수)
- .param (파라미터로 받은 변수)

(4) String 선언 특징
- .field private static final TAG:Ljava/lang/String; = ″
- const-string v3, ″

이후 일반적인 오픈소스의 경우 협업을 위해 낙타 표기법을 따르고 있으므로, 추출한 단어를 기반으로 특수문자들을 삭제하고 문자열을 배열 화한다. 낙타 표기법 관련 추출에 대한 상세한 내용은 Fig. 6. 에 기술하였다. 이후 앱 전체를 하나의 문서로 생각하여 패키지는 디렉토리, 클래스는 제목, 함수는 문단, 변수는 단어로 취급하여 하나의 앱을 문서 묶음으로 보고, 클래스와 메소드를 중심으로 학습한다.

본 제안 기법은 dex 파일을 디컴파일을 이용해 Smali 코드를 사용하므로, 소스 코드 없이 실행 파일만 존재해도 사용할 수 있다.

3.2 모델

온라인 평가를 위해 AI 학습 서버와 평가 서버를 별도로 구성하였고, Class와 Method의 가중치를 강하게 두고 LSTM[14] (Long Short-Term Memory)로 학습하였다. LSTM은 RNN의 정보량 손실에 대한 장기 의존성 문제를 개선하기 위한 것으로, 히든 상태에 셀 상태를 추가한 구조로 되어있다.

- Forget Gate Layer : 정보의 삭제
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
- Input Gate Layer : 정보의 저장
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\widetilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
- Update Cell State : 위 두 결과를 업데이트
$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t$$
- Output Gate Layer : 출력값을 결정
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_0)$$
$$h_t = o_t \odot \tanh(C_t)$$

위 계산 식을 포함하여 LSTM의 네트워크 구조를 도식화하면 Fig. 4.와 같다. 활성 함수는 Sigmoid를 사용하였고, 함수 당 난독화 여부를 참 또는 거짓으로 판별하므로 손실 함수로 Binary crossentropy를 사용하였다.

평가하고자 하는 앱이 Input으로 주어지면, Classifier에서 디컴파일을 한 뒤, Smali 코드를 기반으로 Package, Class, Method, Variable, String을 뽑아내어 분류하고, 휴리스틱 레이어에서 추가로 난독을 판단한다. 이때 3.3장에서 자세히 설

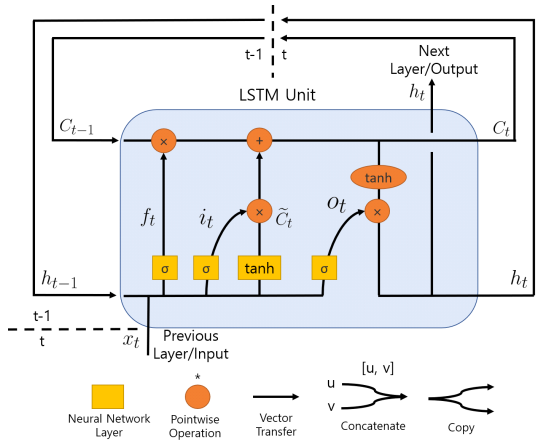


Fig. 4. Structure of LSTM Network

명할 휴리스틱 레이어에서 정규표현식으로 뽑아낸 Class와 Method의 명사와 동사들을 주요 Feature로 사용한다. 이후 Embedding Layer에서는 각 단어를 정제하고, 수치화된 형태로 벡터화한다. 이후 단어 벡터를 LSTM Network를 이용해 학습시키며, 학습된 모델을 통해 각 메소드 단위의 난독화 예측을 진행한다. 이 구조를 도식화하면 Fig. 5.와 같다.

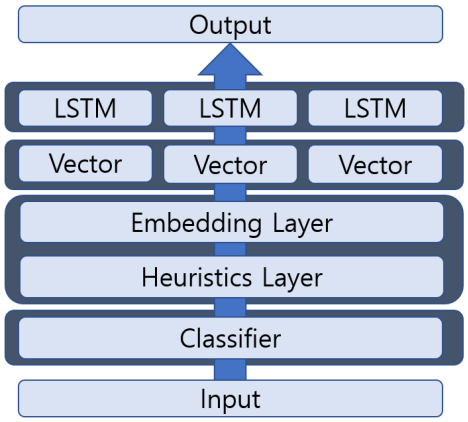


Fig. 5. Structure of LSTMs for AndrObfusec

3.3 휴리스틱 기법

본 연구는 기계학습에만 완전히 의존하지 않고 경험에 기반한 몇 가지 휴리스틱을 사용하여 99%에 달하는 높은 정확도의 학습이 가능하다.

3.3.1 한 글자 난독화

“a”, “b” 등 한 글자로 이루어진 Package, Class, Method는 난독화 된 것으로 간주하였다.

일반적인 개발자는 협업을 위해 Code Convention을 따르므로 Package, Class, Method를 한 글자로 작성하지 않기 때문이다.

3.3.2 Class와 Method의 강한 연관성

실험 시 Package-Class나 Method-Variable보다 Class-Method의 연관성이 중요했다. 반대로 Variable의 경우 Loop 문 등에서 i, j 등의 연관성 없는 단어가 자주 사용되고 외부에서 참조될 가능성이 작아서 이름을 자유롭게 짓는 경우가 발생하므로, 학습에서 배제하였다.

3.3.3 난독화의 중요도

Package - Class - Method - Variable - String 순서로 난독화가 적용된다. 즉, Class는 난독화 되었는데 Method가 난독화 되지 않은 예는 있지만, 그 반대의 경우는 거의 없다. 이것은 Class나 Package가 외부로부터 호출될 가능성이 더 커서 난독화 하기 어려운 경우가 더 많기 때문이다.

특히 String은 결국 어느 지점에서 난독화가 풀린 상태로 존재한다는 점과 기본적으로 제공되는 무료 난독화 툴 ‘Proguard’에서 지원하지 않는다는 점 때문에 다른 경우에 비해 난독화 하지 않는 경우가 더 많았다.

3.3.4 단어 추출

협업을 주로 하는 일반적인 개발자들은 낙타 표기법을 사용하므로, 추출한 단어들은 정규식을 이용한 Fig. 6.의 코드를 통해 특수문자 제거 및 띄어쓰기를 추가하여 전처리하였다. 이를 통해 일부러 낙타 표기법을 사용하지 않고 본인만의 스타일로 코드를 짜는 경우도 난독으로 판단할 수 있게 된다.

```
function camelConverter(str){
    ret = str.replace
       ( /(^[a-z]+)|[0-9]+|[A-Z][a-z]+|[.~!@#$%^&*()_+|<>?:{}]+|[A-Z]+(?=[A-Z][a-z]|[0-9]|[.~!@#$%^&*()_+|<>?:{}])/g
       , function(match, first){
          if (/[.~!@#$%^&*()_+|<>?:{}]/.test(match)) return ' ';
          return match + ' ';
          }
       )
    return ret.replace(/(^ *)|( *$)/g, "").replace(/ +/g, " ");
}
```

Fig. 6. Camel-Case Converter

3.4 난독화도 측정

Class를 C, Method를 M으로 두고 C와 M을 구성하는 단어를 각각 c, m으로 둘 때, 특정 클래스 C에 포함된 M의 난독화도는 다음과 같이 표기한다.

$$C = c_1 c_2 ... c_n$$
$$M = m_1 m_2 ... m_m$$
$$C \bullet M = c_1 c_2 ... c_n m_1 m_2 ... m_m$$
$$C_i = \{M_{i1}, M_{i2}, ..., M_{ij}\}$$
$$n(C_i) = j$$
$$o(x) = \begin{cases} 1 & \text{if } x \text{ is } obfuscated \\ 0 & \text{if } x \text{ is } \neg obfuscated \end{cases}$$

위 조건에서 난독화도 O는 다음과 같이 정의한다.

$$O = \frac{\sum_{x=1}^{i}\sum_{y=1}^{j} o(C_x \bullet M_{xy})}{\sum_{k=1}^{i} n(C_k)} * 100 \quad (M_{ij} \in C_i)$$

여기서 정확도는 AI가 $o(C \bullet M)$을 맞춘 비율로 계산한다.

IV. 실험 결과

4.1 데이터 수집

탐지 기법에 소개한 이유로 오픈소스를 기반으로 16개 카테고리에서 양질의 프로젝트를 수집하였다.

수집에 참고한 프로젝트는 안드로이드 오픈소스 수집 프로젝트 중 가장 유명한 open-source-android-apps[12]이며, 이 프로젝트는 Github에서 개발자들에 의해 8.1k의 Star를 받았고 2.1k회 Fork 되었다. 참고로 이 수치는 가장 유명한 안드로이드 리버싱 툴 중 하나인 dex 2 jar[13]보다 많은 것이다.

Table 3. Crawled Opensource projects

Android TV	2
Android Wear	10
Business	1
Communication	23
Education	12
Finance	8
Game	28
Health and Fitness	6
Life and Style	7
Multimedia	45
News and Magazines	37
Personalization	10
Productivity	30
Social Network	43
Tools	87
Travel and Local	13

Table 4. Parsed Opensource projects

Category	Class	Method	Variable	String
Plain	42748	374160	1336447	86372
Obfuscated	53058	221966	241915	28786

이후 빌드 된 앱을 디컴파일한 뒤 Smali 코드를 순수한 소스 코드와 비교 하는 방식으로 난독화 되지 않은 코드 약 4만 2천여 개와 난독화 된 5만 3천여 개의 Smali 코드를 추출하였다.

4.2 평가

4.2.1 학습

학습 시 99%에 달하는 높은 정확도로 학습할 수 있었다. 여기서 정확도란 예측의 반올림 값이 정답과 일치하는지에 대한 평균값으로, 정확도를 계산하는 코드는 Fig. 7.과 같다.

6회부터는 과적합이 나타나 학습을 종료하도록 했는데, 5회 정도의 단기 학습에도 Fig. 8.과 같이 높은 정확도로 난독화도를 학습할 수 있었다.

```
def binary_accuracy(y_true, y_pred):
    '''Calculates the mean accuracy rate across all predictions for binary
    classification problems.
    '''
    return K.mean(K.equal(y_true, K.round(y_pred)))
```

Fig. 7. Taining and Validation Accuracy

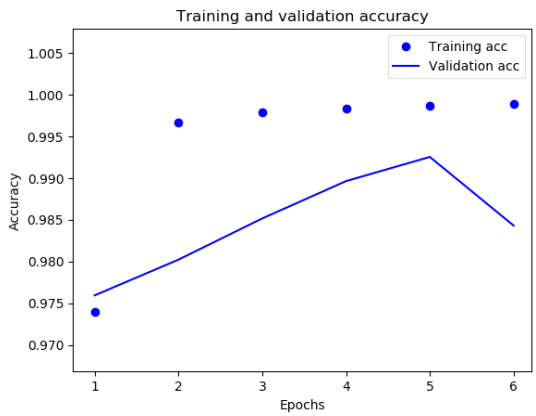


Fig. 8. Taining and Validation Accuracy

Table 5. Obfuscation Coverage

	WhatsApp	INSTAGRAM	Facebook	Youtube	Garena	FIFA	Sample
Accuracy	92.33	89.55	89.29	89.28	95.56	95.51	97.11
O(AI)	79.13	74.81	78.06	84.82	20.26	07.81	02.89
O(Real)	83.89	84.04	86.24	95.22	18.71	07.18	00.00

4.2.2 측정

평가를 위해 상용 앱 중 다운로드, 별점 등 종합 순위 최상위에 랭크된[1] WhatsApp, Instagram, Facebook, Youtube, Garena, FIFA에 대해 50만 개 이상의 클래스, 함수를 평가하여 난독화도와 실제 난독화도를 측정했다. 또한, False Positive에 대한 더욱 정확한 실험을 위해 구글의 안드로이드 전용 개발 환경[2]에서 기본으로 제공하는 샘플 앱을 별도의 난독화 과정 없이 디버그 빌드하여 난독화도를 측정하였다. 측정 결과는 Table 5.와 같다.

측정 결과 상위 앱들은 대체로 난독화가 잘 되어 80%~90%의 높은 난독화도를 나타냄을 알 수 있었고, Garena, FIFA 등 게임 앱의 경우 난독화가 적용은 되어있으나 난독화도가 낮아 제대로 난독화 되어있지 않음을 알 수 있었다. 일반적으로 난독화는 게임 부하에 영향을 미칠 수 있으므로 게임 앱에 많이 적용되지 않은 것으로 보인다.

또한, 전혀 난독화가 되지 않은 샘플 앱에 대해서는 97.11%의 정확도가 나타났으며, 결과에 대한 높은 정확도를 볼 수 있었다.

4.3 논의 및 한계

실험을 통해 학습된 데이터를 이용하여 다음과 같은 흥미로운 사실을 발견할 수 있었다.

Fig. 9. 의 경우, 실제로 "어둠"과 관련된 작업을 하는 클래스이므로 메소드명이 "명도변경", "이벤트" 등으로 구성되어 있다. 그러나 Fig. 10.는 어둠과 관련된 작업을 하지 않지만, 난독화 되어 "어둠"이라는 이름을 갖게 된 미지의 클래스이므로, 연관성이 떨어지는 "코드"라는 메소드로 이루어져 있다.
AndrObfusec은 우리 인간이 이 두 코드를 보고 연관성을 통해 난독화를 구별하는 것처럼 두 코드를 구별하여 올바르게 난독화를 판단하였다.

```
Package Name: meghal developer nightsight project services
Class Name  : Darkness
Method Name : on Brightnesschange Event
```

Fig. 9. Darkness Class (Plain)

```
Pacakge Name: cn bmob push lib service
Class Name  : darkness
Method Name : Code
```

Fig. 10. Darkness Class (Obfuscated)

하지만 만약 서로 의미 있는 단어로 잘 대응하여 인간도 속을 수밖에 없는 난독화 기법이 추후 상용화 된다면, 본 연구로는 한계가 있으며 동적으로 학습하여 탐지하는 기법을 도입하여 해결해야 한다. 예를 들어 네트워크로 패킷을 보내는 행위를 하면 Request 등의 단어와 연관성이 있는 클래스임을 학습하는 식으로 동적 분석과 정적 분석을 동시에 사용해야 한다.

또한, 국가마다 다르게 해석되는 단어(Saram - 한국어로는 사람, 타밀어로는 본질이라는 의미), 특정 국가에서만 쓰이면서도 유사성과 모호성을 가지는 단어(Insa - 서울의 지역, 행동 규범, 기업의 직무), 특정 국가의 고유명사(Wooribank), 신조어나 은어(ZZang)가 있다. 이러한 단어들은 제대로 학습이 되지 않으면 난독으로 판단이 되어 일관성 문제가 생긴다. 그러나 특정 그룹만 알 수 있는 단어는 다른 그룹의 사람들이 쉽게 알기 어려우므로 코드가 읽기 어렵다는 의미인 "난독(難讀)"의 뜻과 일치 한다. 따라서 난독이라고 판단하는 정도가 문화나 국가 등 그룹마다 조금씩 다른 것이 더 자연스럽다.

이후 연구에서는 난독화 정도 분석에 정확도가 비교적 낮은 앱들을 분석하여 그 원인을 파악하고, 속도 문제를 개선하며, 더 많은 오픈소스 앱을 통해 다양한 데이터 세트를 확보하여 더 강인한 탐지 연구를 할 것이다.

1) AndroidRank 선정 기준
2) https://developer.android.com/ 제공

V. 결 론

기존에도 단순히 난독화 여부나 어떤 난독화 툴로 난독화 되었는지 알아보려는 연구는 있었지만 두 가지 문제가 있었다. 첫 번째 문제는 기존에도 난독화 여부를 판단할 때 임곗값을 사용해 난독화 여부를 판단하였으나, 앱 단위로 코드의 유사도나 패턴을 감지하였으므로 난독화의 정도를 세밀하게 파악하기는 어려웠다는 것이고, 두 번째 문제는 사전에 등재된 단어로 되어있으나 내용에 연관성이 없어 사람이 이해하기 힘든 난독화는 판별하기 어려웠다는 점이다.

본 논문에서는 이러한 문제들을 해결하기 위해 앱 단위가 아닌 앱의 클래스와 함수 단위로 묶어 각각의 임곗값을 통해 난독화 여부를 판단하였으므로, 난독화의 정도를 세밀하게 표시할 수 있다. 또한, 자연어 처리를 통해 이해하기 힘든 방식의 난독화를 포함한 난독화도의 측정이 가능하다. 이를 위해 오픈소스 16개 카테고리에서 약 10만 개의 클래스와 60만 개의 함수를 몇 가지 휴리스틱 기법과 함께 연관 지어 99% 정확도로 학습시켰고, 평가 단계에서 약 50만 여 개의 대상에 대해 평균 90% 이상의 높은 정확도를 나타내 난독화도 측정이 가능함을 보였다.

해당 연구를 통해 앱을 얼마나 쉽게 이해할 수 있는지에 대한 정보와 얼마나 보호되어 있는지 정도를 알 수 있게 되었고, 이는 앱 보안성에 대한 간접 지표가 되므로 활용성이 높을 것으로 기대된다.

References

[1] Roedy green. "How To Write Unmaintainable code," *Java Developers' Journal*, Jan. 2000.

[2] "Final Report on Information Security Survey" pp. 59-63. *KISA*, Feb. 2020.

[3] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, Manoj Gaur. "Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions," *arXiv: 1611.10231,* Nov. 2016.

[4] R Mohsen and AM Pinto. "Evaluating obfuscation security: A quantitative approach," *Springer,* vol. 9482, May. 2015.

[5] Atanas Rountev, Yan Wang. "Who Changed You? Obfuscator Identification for Android," *IEEE,* pp. 154-164, July. 2017.

[6] Yasemin Acar, Bradley Reaves, Patrick Traynor, Sascha Fahl, Dominik Wermke, Nicolas Huaman. "A Large Scale Investigation of Obfuscation Use in Google Play," *ACSAC,* Dec. 2018.

[7] J. E. Tapiador, L. Gonz´ales-Manzano, O. Mirzaei, J. M. de Fuentes. AndrODet: An adaptive Android obfuscation detector. *Elsevier,* pp. 222-235, July. 2018.

[8] Nahid Shahmehri, Alireza Mohammadinodooshan, Ulf Karg´en. Comment on "AndrODet: An adaptive Android obfuscation detector". *Future Generation Computer Systems,* pp. 240-261, vol 90, Jan. 2020.

[9] Nahid Shahmehri, Alireza Mohammadinodooshan, Ulf Kargen. "Robust Detection of Obfuscated Strings in Android Apps," *AISec'19,* pp. 25-35, Nov. 2019.

[10] Hyoung-Kee Choi, Dongmin Jo. "Android Application Obfuscation Technique Inducing Misjudgement of Obfuscation Application," *Korea Institute Of Communication Sciences*, 38(8), pp. 654-662, Jan. 2018.

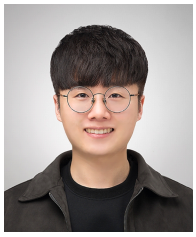
[11] Sun Microsystems. "Java Code Conventions", pp.15-16. *Oracle,* Sep. 1997.

[12] Github, "open-source-android-apps," https://github.com/pcqpcq/open-source-android-apps, May. 2020

[13] Github, "dex2jar," https://github.com/pxb1988/dex2jar, Jun. 2015.

[14] S Hochreiter and J Schmidhuber, "Long short-term memory," *Neural computation*, pp. 1735-1780, Feb. 1997.

〈저 자 소 개〉



김 병 연 (Byeong Yeon Kim) 학생회원
2015년 2월: 단국대학교 소프트웨어학과 학사
2015년 1월~현재: ㈜삼성전자 소프트웨어 엔지니어
2020년 3월~현재: 고려대학교 정보보호대학원 정보보호학과 석사과정
〈관심분야〉 자연어처리, 인공지능보안, 비정상행위탐지, 블록체인

김 휘 강 (Huy Kang Kim) 종신회원
1998년 2월: KAIST 산업경학학과 학사
2000년 2월: KAIST 산업공학과 석사
2009년 2월: KAIST 산업및시스템공학과 박사.
2004년 5월~2010년 2월: 엔씨소프트 정보보안실장. Technical Director.
2010년 3월~2014년 12월: 고려대학교 정보보호대학원 조교수.
2015년 1월~2020년 2월: 고려대학교 정보보호대학원 부교수.
2020년 3월~현재: 고려대학교 정보보호대학원 교수
2020년 3월~현재: 고려대학교 정보보호대학원-삼성SDS 보안 공동연구센터장
〈관심분야〉 온라인게임 보안, 자동차 보안, 침입탐지시스템, 네트워크 보안