Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.30, NO.3, Jun. 2020 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2020.30.3.493

ICFGO : Inter-Procedural Control Flow Graph 난독화를 위한 UI 은닉 및 Dummy Flow 삽입 기법*

심 현 석,[1†] 정 수 환[2‡]
[1,2]숭실대학교(대학원생, 교수)

ICFGO : UI Concealing and Dummy Flow Insertion Method for Inter-Procedural Control Flow Graph Obfuscation*

Hyunseok Shim,[1†] Souhwan Jung[2‡]
[1,2]Soongsil University(Graduate student, Professor)

요 약

안드로이드 운영체제에서 Flow Analysis의 난독화를 위해서는 실행되지 않는 흐름의 코드를 생성하여 Flow Graph의 크기를 크게 만들어 분석이 어렵게 만들 수 있다. 이를 위해 논문에서는 aar 형태의 라이브러리를 구현하여 외부 라이브러리의 형태로 애플리케이션에 삽입이 가능하도록 하였다. 라이브러리는 더미 코드에서의 진입점에서부터 최대 5개의 child node를 가질 수 있도록 설계되었으며, child node의 클래스는 각 node마다 100개부터 900개까지 2n+1개의 메서드를 가지고 있으므로 총 2,500개의 진입점으로 구성된다. 또한 진입점은 XML에서 총 150개의 뷰로 구성되며, 각각의 진입점은 비동기 인터페이스를 통해 연결된다. 따라서 Inter-Procedural 기반의 Control Flow Graph를 생성하는 과정에서는 최대 14,175E+11개의 추가적인 경우의 수를 가지게 된다. 이를 애플리케이션에 적용한 결과 Inter Procedural Control Flow Analysis 툴에서 평균 10,931개의 Edge와 3,015개의 Node가 추가 생성되었으며 평균 36.64%의 그래프 크기 증가율을 갖는다. 또한 APK를 분석 시에는 최대 평균 76.33MB의 오버헤드가 발생하였지만, 사용자의 ART 환경에서는 최대 평균 0.88MB의 실행 오버헤드만을 가지며 실행 가능한 것을 확인하였다.

ABSTRACT

For the obfuscation of Flow Analysis on the Android operating system, the size of the Flow Graph can be large enough to make analysis difficult. To this end, a library in the form of aar was implemented so that it could be inserted into the application in the form of an external library. The library is designed to have up to five child nodes from the entry point in the dummy code, and for each depth has 2n+1 numbers of methods from 100 to 900 for each node, so it consists of a total of 2,500 entry points. In addition, entry points consist of a total of 150 views in XML, each of which is connected via asynchronous interface. Thus, the process of creating a Inter-procedural Control Flow Graph has a maximum of 14,175E+11 additional cases. As a result of applying this to application, the Inter Procedure Control Flow Analysis tool

Received(01. 13 2020), Modified(1st: 03. 02. 2020, 2nd: 04. 14. 2020), Accepted(04. 14. 2020)
* 본 논문은 2019년도 한국정보보호학회 동계학술대회에 발표한 우수논문을 개선 및 확장한 것임
* 이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2020-0-00952, 5G+ 서비스 안정성 보장을 위한 엣지 시큐리티 기술 개발) 또한 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2019-0-00477, 가상화된 신뢰실행환경을 이용한 안드로이드 보안 프레임워크 기술 개발)
† 주저자, ant_tree@naver.com
‡ 교신저자, souhwanj@ssu.ac.kr(Corresponding author)

generates an average of 10,931 edges and 3,015 nodes with an average graph size increase of 36.64%. In addition, in the APK analyzing process showed that up to average 76.33MB of overhead, but only 0.88MB of execution overhead in the user's ART environment.

Keywords: Android application, Flow analysis, Obfuscation

I. Introduction

안드로이드 운영체제에서 개인정보 유출을 분석하는 방법은 정적분석과 동적 분석 방법이 있으며, 이 중 정적분석 방법에서는 흐름 분석을 통한 방법이 주된 방법으로 제시되고 있다. 흐름 분석은 각 함수 호출을 비롯한 실행 흐름을 알아내기 위한 분석 방법으로, 제어 흐름 분석(Control Flow Analysis)이나 데이터 흐름 분석(Data Flow Analysis)과 같이 여러 종류를 가지고 있다. 안드로이드 애플리케이션은 그 특성상 사용자와의 인터랙션(interaction)을 담당하는 뷰를 통해 구성되며, 이러한 인터랙션은 클릭, 터치와 같은 사용자 입력에 의해 이루어진다. 한편 각 뷰는 각각의 진입점(entry point)으로 여러 개의 프로시져(procedure)를 가지므로, Control Flow Graph(CFG)를 생성할 때 이를 고려한 Inter-Procedural 기반 CFG (ICFG)를 생성하고 분석하는 데에 초점을 맞추고 있다. ICFG는 UI 컴포넌트에서의 이벤트나, *broadcastReceiver*와 같은 비동기 지점에서의 흐름을 분석하는 장점을 가지고 있지만, 현재 ICFG 분석 툴에서는 UI 컴포넌트에 대한 면밀한 분석이 이루어지지 않는다. 기존에 존재하는 툴에서는 XML의 분석을 통한 UI 컴포넌트의 분석이 이루어지지만 이는 UI 컴포넌트에 대한 존재 여부를 파악하기 위함이다. 따라서 기존의 툴에서는 inter-procedural 환경에서 진입점의 존재 여부만을 확인하며 해당 진입점의 진입 가능 여부는 파악이 불가능하다. 반면 UI 컴포넌트 상에서의 사용자 접근 가능 여부를 확인하는 과정이 이루어지지 않는다면 정적분석 환경에서는 모든 entry point로의 진입이 유효한 것으로 파악하게 된다. 우리는 이러한 점을 이용하여 ICFG에 대한 난독화가 가능한 것을 확인하였으며, 이를 제어 흐름 그래프 난독화 툴인 **ICFGO**로 구현하였다.

ICFGO는 하드 코딩된 리소스를 포함하여 150개의 entry point와 총 2,500개의 dummy flow를 가진다. **ICFGO**는 안드로이드 런타임(ART)에서는 이러한 흐름을 실행시키지 않지만, 정적분석 환경에서는 해당 코드들의 흐름이 탐지된다.

구현을 위한 기법은 UI 컴포넌트를 배치한 후 은닉하는 것으로, 일반 사용자는 실행이 불가능하지만, 코드상에서는 존재하는 흐름을 만드는 것이다. APK는 UI 컴포넌트를 통한 수많은 entry point를 가지게 되지만, 이러한 UI 컴포넌트는 크기가 0px이거나 화면 밖에 배치되므로 실행이 불가능하다. 이러한 흐름을 통해 APK는 실행 상에서 최소한의 오버헤드를 가지며 실행되는 반면, 분석가의 관점에서는 2차, 3차적인 분석이 불가능할 정도로 큰 그래프를 가지게 된다.

논문은 다음 장부터 배경지식을 설명하며, 3장에서 현재 흐름 분석의 문제점에 대해 설명할 것이다. 이후 4장에서 논문에서의 툴을 구현하는 방법을 소개하며, 5장에서 툴에 대한 평가가 이루어진다. 논문은 마지막으로 6장에서 결론을 통해 끝맺는다.

II. Background

2.1 Android Application

안드로이드 애플리케이션은 디컴파일 시 7개의 주요 경로 및 파일로 나타난다. 이는 classes.dex, res, lib, assets, META-INF, resources.arsc, 그리고 AndroidManifest.xml로, 각각은 디컴파일 툴을 통하여 거의 원본과 동일하게 추출이 가능하다. 흐름 분석을 포함한 일반적인 안드로이드의 정적분석 과정에서는 이 중 AndroidManifest.xml와 classes.dex의 정보를 주로 활용한다[1]. 반면 뷰의 존재 여부 파악을 위한 분석을 병행하기 위해서는 레이아웃 및 drawable 들에 대한 분석이 함께 이루어져야 한다.

안드로이드 애플리케이션은 뷰를 생성하기 위해 리소스 파일 중 하나인 XML 파일에 의존하는데, 이는 res/ 디렉토리의 layout/ 디렉토리에 위치한다. 각각의 뷰 구성요소는 뷰 내부의 리소스 배치를 위하여 이미지 리소스에 접근하며, 이는 res/ 디렉터리의 drawable-*/과 mipmap-*/ 경로에 포함되어 있다.

resources.arsc 파일에는 string, layout의 정

보를 포함하는 id를 가지는 모든 요소들에 대해 각 리소스와 해당 리소스의 id의 매핑이 이루어진다[2]. 따라서 arsc 파일에 정의된 id와 레이아웃 상에 정의된 뷰 id를 매칭하여, 각각 레이아웃에서 사용되는 뷰에 대한 리소스 id를 추출 가능하다. 이러한 방식은 레이아웃 및 drawable에 대한 존재 여부 파악 및 분석에 사용되므로, 뷰를 포함한 분석에 있어 필수적인 요소이다.

2.2 Control Flow Graph

Control Flow Graph(CFG)는 각 기본 블록(basic block)으로 구성된 절차적 그래프로, 각 노드는 statement에 대응된다. CFG의 목적은 실행 가능한 모든 경로를 알아내는 것으로, 모든 경로는 statement에 대응하는 노드와 실행 경로를 의미하는 edge로 구성된다.

안드로이드 운영체제에서 대부분의 CFG는 Inter-procedural Control Flow Graph(ICFG)와 Callback Control Flow Graph(CCFG)로 나타난다. 고전적인 ICFG는 Fig. 1과 같이 main procedure에서의 진입점에 의존하지만, CCFG에서는 사용자가 실행하는 환경에서의 이벤트와 이벤트 핸들러에 의해 나타난다[3]. 안드로이드에서의 CFG 생성은 프레임워크 기반의 특징과, 이벤트에 의해 작동하는 특징으로 인해 ICFG 보다 CCFG가 더 적합하다[4]. 반면, 안드로이드에서는 CFG를 시작하는 main procedure가 존재하지 않는데, 이런 문제를 해결하기 위해 Steven Arzt et al. 은 **FlowDroid**[5]와 같은 오픈소스 툴을 개발하였다.

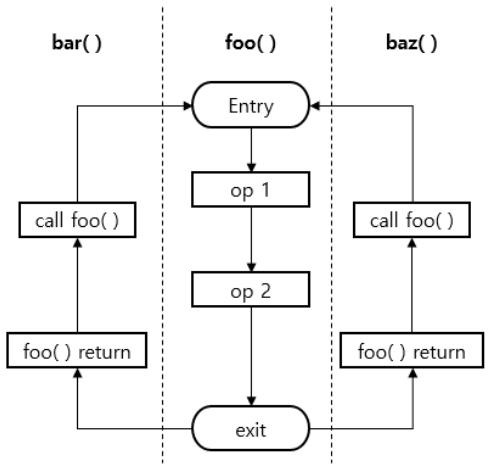

Fig. 1. Example of inter-procedural control flow graph. This graph is consists of 1 main procedure and 2 extra procedures.

2.3 Obfuscation

안드로이드 어플리케이션에 대한 난독화 기법은 이미 여러 종류가 소개된 바 있다[6][7][8]. 이러한 난독화 기법들 중 **Proguard**와 같은 툴은 안드로이드에서 기본 제공하는 난독화 툴 중 하나로, 클래스와 패키지 및 멤버 변수에 대한 이름 축소 및 난독화를 수행한다. 또한 **Dexguard**와 같은 난독화 툴은 classes.dex 파일에 대한 **Proguard**와 동일한 난독화를 수행하며, 더불어서 리소스 파일들에 대한 난독화를 수행한다. 그 결과 apk 파일 내부의 리소스 파일들의 난독화까지 수행되며, 일반적인 난독화 툴과는 다르게 디컴파일 결과에서 리소스에 대한 접근이 불가능하다. 안드로이드 난독화 툴 중 일부는 제어 흐름 분석에 대한 난독화를 수행하며[9][10], 그 기법은 대부분 control flow flattening을 통해 수행된다.

III. Related work

최근 ICFG 분석을 수행하는 수많은 툴들이 연구되어왔으며, 논문을 통해 공개되었다. 그러나 이러한 작업들은 진입점의 진입 가능여부를 판단하지 않으며, 최대한 많은 진입점을 찾는 것에 초점을 두고 있다. **FlowDroid**는 *dummyMainMethod*를 이용하여 여러 진입점을 가지는 문제와 비동기로 실행되는 코드 및 *callback*까지 분석하는 툴이다. **FlowDroid**에서는 inter-component 분석을 위해 XML에서의 뷰 컴포넌트를 분석하며, 어떤 뷰가 어떤 코드로 연결되었는지 파악한다. 따라서 각 *button* 혹은 *textView*와 같은 뷰에서 연결되는 진입점을 찾는 것은 가능하지만, 이러한 진입점이 실제로 user-level에서 진입 가능한지 확인하는 부분은 누락되어있다. **IccTA**[11]는 안드로이드 운영체제에서 inter-component 간의 흐름을 포함한 flow analysis를 수행하기 위한 툴로, lifecycle 메서드와 콜백 메서드를 분석하는 기능을 포함한다. **IccTA**에서는 *onClick*과 같은 뷰에서 전달되는 콜백 메서드의 흐름을 추적하여 분석하지만, 해당 뷰와

Table 1. Comparison of tools for analyzing on ICFG (acc. = accessibility)

Tool	ICC analysis	View analysis	acc. analysis
FlowDroid	O	O	X
IccTA	O	O	X
AmanDroid	O	O	X
DroidRista	O	X	X

연결된 java 코드에서만 콜백 메서드를 분석하므로 뷰의 접근 가능 여부는 확인하지 않는다. **AmanDroid**[12]는 Inter/intra-component 간의 Data Flow Graph(DFG)와 CFG를 모두 지원하며, XML 리소스 기반의 분석 또한 가능하다. 반면 XML 리소스에서 특정 뷰의 접근 가능 여부를 파악하지 않으므로 접근 가능하지 않은 뷰에서는 false positive를 가진다. **DroidRista**[13]는 ICC 메서드들에 대한 분석을 수행하며, activity 전환과 같은 메서드들에 대한 context-sensitive한 분석을 수행하지만, 뷰 요소를 통한 비동기 흐름에 대한 분석은 이루어지지 않는다. 이러한 기존 툴에서의 분석 범위에 대한 비교는 Table 1을 통해 나타내었다. 논문에서 제안하는 툴은 이러한 기존 툴의 분석 문제점을 활용하여 난독화를 진행하기 위함이다. 따라서 논문의 툴은 ICFG 생성 단계에서 진입 가능성을 고려하지 않는 분석 툴 들을 타겟으로 하며, 이는 현재 주류를 이루는 거의 모든 CFG 분석 툴이라 할 수 있다. 다음 섹션부터 구현 방법에 대한 상세한 기술이 이루어진다.

IV. Implementation

4.1 Overview

본 난독화 툴의 전반적인 구조는 Fig. 2에 표시된 것과 같다. ICFGO는 원본 소스를 가진 개발자가 해당 툴을 통해 난독화가 가능하도록 설계되었으며, 원본 소스코드에 라이브러리를 추가하여 더미 코드를 추가할 수 있다. 라이브러리의 메인 함수에서는 개발자로부터 타겟 액티비티와 복잡도, 그리고 최상위 뷰의 ID를 입력받으며, 이를 통해 XML 혹은 동적 뷰 inflating을 통해 접근 불가능한 뷰를 생성한다. 이 후 각각의 뷰에 대한 인터페이스 선언을 통해 dummy entry point를 생성하며, 각 진입점에서

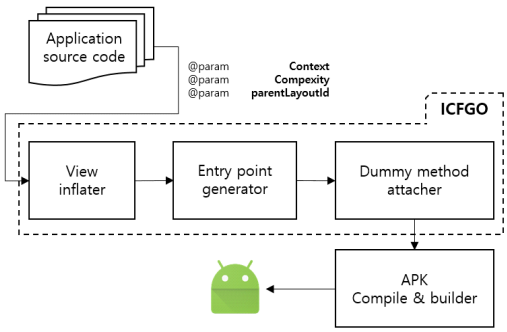

Fig. 2. Overall architecture and obfuscation pipeline of ICFGO

부터 최대 depth까지 dummy method를 연결하는 구조로 이루어진다. 이후 개발자는 dummy flow가 생성된 소스를 통해 APK를 빌드할 수 있다. 각 단계에서의 흐름은 4.2부터 기술하였다.

4.2 Dummy View Insertion

본 논문에서는 dummy view를 삽입하기 위해 약 150개의 dummy view를 가지는 XML 코드와 java 코드 내에서 동적으로 뷰를 추가하는 기능을 구현하였다. Dummy view를 포함하는 XML의 경우, *focusable=false* 혹은 *clickable=false* 등을 이용하여 뷰에 이벤트를 발생시키는 것이 불가능하게 하거나, 혹은 *visibility=gone*으로 정의하거나 width와 height가 0px로 정의하여 뷰에 대한 접근이 불가능하도록 숨겨놓았다. 또한 java 코드에서는 개발자가 지정한 complexity에 따라 XML의 뷰를 inflating하는 기능을 가지고 있으며, 만약 complexity가 XML에서 정의된 최대 개수인 150개를 초과한다면 동적으로 뷰를 추가한다. 뷰의 동적 추가 과정은 하나의 ViewGroup에 complexity에 대응하는 수만큼의 뷰를 추가하고, 이를 *visibility=gone*으로 설정하여 숨기는 방식으로 이루어진다. 이러한 코드의 흐름을 Fig. 3을 통해 나타내었다.

4.3 Dummy Entry Point Insertion

이 섹션에서는 각 뷰에 대해 임의의 이벤트 리스너를 부착하여 랜덤으로 entry point를 생성하는 과정에 대해 소개한다. 각 뷰는 *AnonymousInterf*

```
1  LayoutInflater inflater = (LayoutInflater) context
   .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
2  ...
3  if (complexity > 50) {
4      viewExtender(context, inflater,
5      parentLayout, R.layout.view_item_iv50a);
6  }
7  ...
8  if (complexity > 150) {
9      ...
10     for (int viewCount = 0; viewCount <
       extraComplexity; viewCount++) {
11         View view = new View(context);
12         view.setVisibility(View.GONE);
13         frameLayout.addView(view);
14     }
15     parentLayout.addView(frameLayout);
16 }
```

```
<TextView
    android:id="@+id/txt_t30b20a_b1"
    android:layout_width="0dp"
    android:layout_height="0dp" />
<Button
    android:id="@+id/btn_t30b20a_b2"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:focusable="false" />
```

Fig. 3. (Top) Java codes to inflate views from XML or extend dynamically according to complexity (Bottom) XML codes for UI conceal.

*ace*로 정의되는 custom interface를 가지고 있으며, 각각은 4단계로 이루어진 depth에 따라 임의의 그래프 흐름을 생성한다. Fig. 4는 랜덤으로 생성된

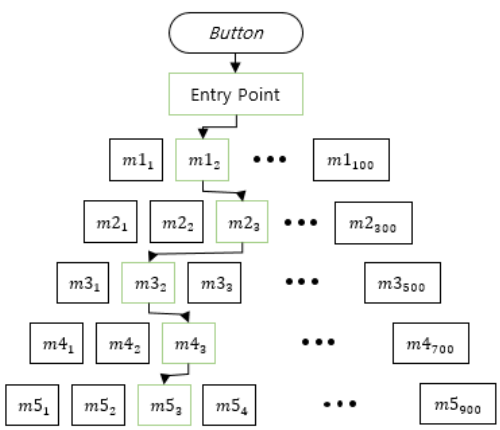

Fig. 4. Possible dummy flow from dummy entry point to real entry point

dummy entry point와 해당 entry point에서 이어질 수 있는 가능한 흐름을 나타낸 것이다. 그림에서 각 depth에 따라 생성 가능한 흐름은 단계별로 100개, 300개, 500개, 700개, 900개로, 총 14,175E+11개의 경우의 수를 가지며, 이는 각 진입점으로부터 4단계의 depth를 가지는 서브그래프로 나타난다.

각 진입점에서 연결되는 실행 흐름은 추상클래스로 구성된 메서드를 랜덤으로 호출하며 생성된다. 이

```
ALGORITHM 1: Creating Dummy Flow
    Input: C : complexity.
    Input: maxC : number of views in XML.
    Input: maxD : maximum depth of dummy flow.
    Output: dummyFlow : dummy flow of control flow graph.
    Variable: dummyViews : set of dummy views inflated by XML.
    Variable: dynamicDummyView : set of dummy views inflated dynamically.
    Variable: DEP : Dummy enty point.
 1  procedure InflateDummyView(C)
 2      IF C is smaller than given complexity C DO
 3          dummyViews <- inflateXMLs(C);
 4      ELSE
 5          dummyViews <- inflateXMLs(maxC);
 6          extraC <- subtract(C, maxC);
 7          currentC <- extraC;
 8          WHILE currentC is not reached extraC DO
 9              dynamicDummyViews <- inflateDynamic(currentC);
10              re-compute current complexity;
11          setObfuscation(dynamicDummyViews);
12      setObfuscation(dummyViews);
13  procedure setObfuscation(v)
14      FOREACH view of set of views v DO
15          aInterface <- get anonymous interface;
16          DEP <- attachListener(view);
17          setDummyFlow(DEPS);
18  procedure setDummyFlow(DEPS)
19      FOREACH dummy entry point DEP of set of entry point DEPS DO
20          dummyFlow <- Ø;
21          SETcurrentDepth 0;
22          dummyFlow <- dummyFlow ∪ setDummyFlow(DEP,currentDepth);
23  procedure setDummyFlow(DEP, depth)
24      dummyFlow<- Ø;
25      WHILE depth is smaller than maxD DO
26          dummyMethods <- dummyMethods ∪ dummy methods of depth;
27          assign one of dummyMethods to DEP randomly;
28          re-compute currentDepth;
29      dummyFlow <- DEP with dummyMethods;
30      return dummyFlow;
```

Fig. 5. Algorithm for creating dummy flow from dummy entry point

를 위해 *onClickListener*, *onLongClickListener*, *onTouchListener*를 랜덤으로 반환하는 인터페이스인 *AnonymousInterface*로부터 5단계의 클래스 호출이 이루어진다. 그래프에서 노드에 해당되는 요소인 뷰와 dummy entry point를 각각 $v \in V$, $e \in E$ 로 정의하며, dummy method를 m∈M로 정의하였을 때 대응되는 서브그래프는 $subgraph(v) \subseteq e \rightarrow m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5$로 나타난다. 따라서 CFG를 생성하는 과정에서 노드는 진입점 혹은 dummy method를 의미한다. 전체 생성 과정에 대한 pseudo code는 Fig. 5와 같이 나타난다.

4.4 Library Extraction

본 논문에서 개발된 툴은 aar 형태의 라이브러리로 추출되어 개발자의 gradle 환경에서 import 및 implementetion이 가능하도록 한다. aar 라이브러리는 15이상 28이하의 SDK level을 타겟으로 하였으며, 각 리소스 및 레이아웃을 포함하고 있어 single line으로 사용이 가능하도록 하였다. 라이브러리는 구현 위치의 Activity나 Fragment, 혹은 공통으로 사용되는 Base Context 상에서 라이프사이클의 첫 번째 순서인 *onCreate*()에서 호출하여 사용한다. 일반적으로 안드로이드 어플리케이션에서는 *onCreate*() 시점에서 뷰에 대한 초기화 과정이 이루어지기 때문에, 해당 순서에서 dummy view의 초기화가 이루어지는 것이 올바르다. aar 형태의 라이브러리는 개발자로 하여금 기능의 이식이 쉽지만, 난독화 툴의 측면에서 보았을 때는 개별 패키지의 형태로 존재하기 때문에 난독화 여부를 식별하기 쉽다는 단점을 가진다. 따라서 ICFGO는 패키지 및 클래스의 이름을 난독화하는 다른 툴과 병행하는 것이 올바르다.

또한 aar 라이브러리의 사용을 위해서는 ICFGO의 dummy flow가 삭제되지 않기 위해 리소스 축소와 같은 기능을 사용해서는 안 되며, 출시를 위한 apk를 추출할 때 minifyEnable 속성을 false로 두어야 한다.

V. Evaluation

본 논문에서는 뷰의 개수에 따른 난독화 여부의 확인을 위해 Activity를 비롯한 레이아웃 개수를 변화시켜가며 성능 평가를 진행하였다. 또한 실제 서비스 중인 애플리케이션에 해당 라이브러리를 적용하여 적용 전후의 성능을 평가하여 Table 2와 Table 3에 나타내었다. 성능 평가 지표는 애플리케이션을 실행하는 환경에서의 메모리 오버헤드와 분석 시 메모리 오버헤드, 그리고 그래프 간의 차이로 구성된다.

그래프 간의 차이를 계산하기 위해 Graph Edit Distance(GED)를 사용하였다. 일반적으로 GED는 두 그래프 G, G′에 대해 G를 G′으로 만들기 위해 필요한 비용으로 나타낸다. 이 과정에서 추가되고 삭제된 에지를 각각 E_I, E_D로, G, G′의 모든 노드를 V, V′로 정의하면 GED(G, G′)는 식 1과 같이 나타난다[14].

$$GED(G,G') = \min(\sum(v_I) + \sum(v_D) + |E_I| + |E_D|),\ (v_I \in \{V'-V\},\ v_D \in \{V-V'\}) \quad (1)$$

식 1에 따라, 두 그래프에서의 생성을 위한 비용

Table 2. Graph Edit Distance based performance Evaluation

# of activities	Original			ICFGO applied			GED(Ø, G) - GED(Ø, G')	Increase ratio
	Graph Nodes	Graph Edges	GED (Ø, G)	Graph Nodes	Graph Edges	GED (Ø, G)		
Single activity	2,336	9,242	11,578	5,084	13,092	18,176	6,598	56.98
2 activities	2,311	9,149	11,460	4,995	13,026	18,021	6,561	57.25
3 activities	2,688	9,434	12,122	5,336	14,088	19,424	7,302	60.23
11 activities (ServiceApp)	8,450	45,494	53,944	11,832	54,443	66,275	12,331	22.85
15 activities (ServiceApp)	10,319	139,472	149,791	13,473	153,158	166,631	16,840	11.24
21 activities (ServiceApp)	17,380	283,991	301,371	20,856	314,562	335,418	34,047	11.29

Table 3. Memory Consumption based Obfuscation Performance Evaluation (max.mem.cons. = maximum memory consumption), (R* = Running, A* = Analyzing)

# of activities	Original		ICFGO applied		R* Overhead (MB)	A* Overhead (MB)
	R*max.mem. cons.(MB)	A*max.mem. cons.(MB)	R*max.mem. cons.(MB)	A*max.mem. cons.(MB)		
Single activity	3.06	1,456	3.98	1,486	0.94	129
2 activities	3.92	1,524	4.54	1,472	0.66	28
3 activities	3.76	1,585	4.01	1,602	0.25	17
11 activities (ServiceApp)	10.06	1,633	11.10	1,672	1.04	95
15 activities (ServiceApp)	7.52	1,859	8.79	1,992	1.27	133
21 activities (ServiceApp)	11.21	2,083	12.32	2,139	1.11	56

차이는 GED(Ø, G)와 GED(G', Ø)의 차이로 나타낼 수 있다. 우리는 FlowDroid를 이용하여 테스트를 진행하였다. 결과에 의하면 모든 난독화 과정은 같은 complexity를 통해 수행되며, FlowDroid의 특성 상 XML 코드의 경우 단 한번의 분석만을 진행하기 때문에 activity의 개수에는 무관한 특징을 보였다. 이로 인해 크게 다르지 않은 GED 차이를 가지므로, 레이아웃의 수가 적은 애플리케이션에서 더 높은 GED 증가율을 가지는 것을 확인하였다. 또한 실행 루틴이 같더라도 기기가 소모하는 메모리는 다르게 나타나므로, 정확한 최대 메모리 소모량을 측정하는데 어려움이 있다. 결과에서는 실행 중 오버헤드의 증가 비율은 높지만, 실제로 발생하는 오버헤드는 1.3MB 미만으로 적은 메모리 증가율을 보인다.

반면 다른 적용 사례를 확인하였을 때 의도와는 다르게 그래프의 크기 증가율이 크게 나타나지 않는 경우를 확인하였다. 이러한 경우는 대부분 사용되지 않는 리소스를 제거하는 shrink resource 기능이 활성화 된 것으로, context를 공유하지 않는 기능들에 대해 사용되지 않는다고 판단되는 것을 삭제한 결과이다.

VI. Limitation

논문에서의 평가는 state-of-art CFG 툴인 **FlowDroid**만을 통해 이루어졌지만, **FlowDroid**는 현재 여러 문제점이 지적된 바 있다. **FlowDroid**와 그 바탕이 되는 **Soot**[15]는 지속적인 업데이트를 거치고 있지만, 여전히 평가 툴 자체에서의 분석 실패율이 평가된 성능에 일부 영향을 미칠 가능성이 있다. 또한 논문의 Evaluation 섹션에는 기재하지 않았지만, 오히려 Inter-procedure 간의 분석을 수행하지 않거나, 콜백에 대한 분석을 수행하지 않는 원시적인 툴에서는 난독화에 실패하는 것을 확인할 수 있었다. 이는 UI 은닉 기법을 기반으로 수행한 Entry Point를 탐지하지 못하기 때문이며, 따라서 이러한 진입가능성 여부를 판단하기 이전에 해당 경로를 무시하기 때문이다.

논문에서 제안한 방법을 구현한 툴은, 그 dummy 메서드가 충분하게 복잡하게 구현되지 않아 단순한 기능만을 수행하는 dummy 메서드임을 확인할 수 있는 여지가 있다. 이를 위해 각 메서드는 더 복잡한 기능셋을 가지고 있을 필요가 있다. 또한 기존의 안드로이드 분석 툴에서 별도의 모듈을 통해 진입 가능 여부를 파악하는 경우 난독화 성능이 저하될 수 있는데, 이러한 경우에는 뷰임을 예측하기 어려운 커스텀 뷰를 생성하여 XML에 정의함으로서 대응이 가능하다.

VII. Conclusion

본 논문에서는 관련 연구들에서 다루지 않은 뷰 및 entry point에 대한 접근 가능 여부 파악에 대한 취약점을 파악하여 이를 이용한 ICFG 난독화를 진행하였다. 개발된 툴인 ICFGO의 실행파일은 **Github** (https://github.com/Anttree1563/ICFGO.git)를 통해 공개되었으며, 모든 사용자가 수정, 배포할 수 있도록 하였다.

이 연구는 ICFG에 대한 난독화가 가능한 것뿐만이 아니라, 실제로 ICFG 생성 툴에서도 false

positive를 줄이기 위한 XML 분석이 필요함을 나타낸다. ICFG의 난독화를 수행하고자 하는 악성코드의 수는 많지 않을 것으로 예상되며, 따라서 이러한 패턴을 분석하는 것은 중요한 의미를 가지지 않을 수 있다. 그러나 XML 분석을 수행한다면, ICFG 생성 툴에서 생성 단계의 접근이 불가능한 UI 컴포넌트와 그러한 컴포넌트와 연결된 실행 불가능한 흐름을 제거하는 것이 가능하며, 이는 분석 시 false positive를 줄이는 것에 기여할 수 있다. 또한 해당 분석 방법으로는 반대로 은닉된 뷰를 탐색하여 해당 흐름만을 들여다 보는 것이 가능하다. 이는 숨겨진 UI 요소를 통해 악성 행위를 수행하는 adware와 같은 유형을 탐지하는 데에 효과적일 것이다. 이러한 이유로 인해 향후 연구로는 XML 분석과 이를 이용한 ICFG 생성의 고도화에 대한 연구가 이루어 질 것이다.

References

[1] N. Peiravian and X. Zhu, "Machine Learning for Android Malware Detection Using Permission and API Calls", 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, pp. 300-305, Nov. 2013.

[2] Suleiman Y. Yerima, Sakir Sezer, Gavin McWilliams, Igor Muttik. "A New Android Malware Detection Approach Using Bayesian Classification", 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), pp. 121-128, Mar. 2013.

[3] A. Kamil, J. Su and K. Yelick, "Making Sequential Consistency Practical in Titanium," Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, pp. 15-15, Nov. 2005

[4] Noriyuki, S., Tetsuo, K., Katsuhisa, M., "Detecting Invalid Layer Combinations Using Control-Flow Analysis for Android," in Proc. COP'16 Proceedings of the 8th International Workshop on Context-Oriented Programming, Rome, pp. 27-32, Jul. 2016.

[5] A. Steven, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps.", Acm Sigplan Notices, vol.49, no.6 pp. 259-269, Jun. 2014.

[6] GuardSquare, "Dexguard", https://www.guardsquare.com/en/products/dexguard, Last Accessed 08 Jan. 2020.

[7] Android Developers, "Proguard", https://developer.android.com/studio/build/shrink-code?hl=ko, Last Accessed 08 Jan. 2020

[8] Allatori Android Obfuscator, "Android Obfuscation - Java Obfuscator", http://www.allatori.com/features/android-obfuscation.html, Last Accessed 08 Jan. 2020.

[9] Vivek Blanchandran, Sufatrio, Darell J.J. Tan, Vrizlynn L.L. Thing, "Control flow obfuscation for Android Applications," Computer & Security Elsevier, vol.61, pp. 72-93, May. 2016

[10] Yong Peng, Jie Liang, Qi Li, "A control flow obfuscation method for Android applications," 2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS), pp. 94-98, Aug. 2016.

[11] L. Li, et al, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps", 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 280-291, May. 2015.

[12] F. Wei, et al, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps", ACM Transactions on Privacy and Security, vol.21, no.3, pp. 1-32, Apr. 2018.

[13] Areej Alzaidi, Suhair Alshehri, Seyed

M. Buhari, "DroidRista: a highly precise static data flow analysis framework for android applications," International Journal of Information Security (2019), pp. 1-14, Oct. 2019.
[14] Zhang, M., and Duan, Y., "Semantics-aware android malware classification using weighted contextual api dependency graphs,", CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1105-1116, Nov. 2014.
[15] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren et al. "Soot: A java bytecode optimization framework.", In CASCON First Decade High Impact Papers, pp. 214-224, Nov. 2010.

〈저자소개〉

심 현 석 (Hyunseok Shim) 학생회원
2019년 2월: 숭실대학교 전자정보공학과 졸업
2019년 3월~현재: 숭실대학교 정보통신공학과 석사과정
〈관심분야〉 모바일 보안, 제어 흐름 분석, 개인정보 보호

정 수 환 (Souhwan Jung) 종신회원
1985년 2월: 서울대학교 전자공학과 졸업
1987년 2월: 서울대학교 전자공학과 석사
1996년 6월: University of Washington 박사
1988년~1991년: 한국통신 전임 연구원
1997년~현재: 숭실대학교 전자정보공학부 교수
〈관심분야〉 AI 보안, 모바일 보안, 클라우드 보안, 네트워크 보안