강화된 난독화 기법을 통한 안드로이드 애플리케이션의 보안 강화

이동호 | 숭실대학교
조해현 | 숭실대학교
손기욱 | 서울과학기술대학교

| 목차

모바일 기기 사용률이 급증함에 따라 개인 정보 유출 등 보안 위협이 발생하고 있다. 안드로이드 애플리케이션의 역난독화 기법이 악성 행위를 하는 공격자들에 의해 사용되면서 점점 더 교묘해지고 있어 사이버 안보에 심각한 위협이 되고 있다. 사이버 보안 위협에 대응하기 위해 안드로이드 애플리케이션을 보호하기 위한 강화된 난독화 기법이 요구된다. 본 연구는 네 가지 난독화 기법을 접목한 강화된 난독화 기술로 정적 및 동적 역난독화 기술들에 대해 저항력을 얼마나 향상하는지에 대한 실증적 검증을 진행한다. 제안 기법은 Sensitive Code Obfuscation, Screen Element Protection, Android Data Storage Space Utilization, Source Code Obfuscation 기법을 토대로 자동화 동적 분석 도구, 역난독화 도구에 대한 방어 전략을 제공한다. 연구 결과는 제안된 난독화 기법이 기존 난독화 방법에 비해 분석률이 현저히 낮으며, 자동화 분석 도구에 대한 보안 강화에 있어서 상당한 이점을 제공한다. 이러한 결과는 안드로이드 애플리케이션 보안 강화에 중요한 기여를 할 것으로 기대한다.

| 주제어 리버스 엔지니어링, 안드로이드, 난독화, 역난독화, 정적분석, 동적분석, 정적오염분석

* 본 연구는 고려대 암호기술 특화연구센터 (UD210027XD)를 통한 방위사업청과 국방과학 연구소의 연구비 지원으로 수행되었습니다.

Ⅰ. 서론

현대 사회에서 모바일 기기의 보급률은 증가하고 있으며 사이버 보안의 중요성도 점차 증가하고 있다. 특히, 안드로이드 플랫폼을 기반으로 하는 애플리케이션들은 그 개방성과 유연성 때문에 널리 사용되고 있지만 이러한 특성은 다양한 보안 취약점을 노출하는 원인이 된다. 최근 안드로이드 애플리케이션을 통해 개인정보가 유출되는 등 보안 이슈가 지속해서 제기되고 있다(김영명 2024).

이에 따라 애플리케이션 난독화는 공격자들의 역난독화 시도를 방해하여 보안을 강화하는 효과적인 방법으로 주목받고 있다(Aonzo et al. 2020). 그러나 기존의 난독화 기법들은 지속해서 발전하는 역난독화 기술에 대응하기에는 한계가 있다. 본 연구는 기존의 난독화 방법을 개선하고 역난독화 공격에 더욱 효과적으로 대응할 수 있는 강력한 난독화 기법을 제안한다.

제안된 기술은 자동화된 분석 도구들과 정·동적 분석에 대한 저항력을 향상하는 것을 목표로 한다. 본 연구에서 제안하는 강화된 난독화 방법은 서로 다른 보호 영역 공간을 접목한 기술로서 다음 네 가지 기법으로 구성되어 있다.

첫 번째 기법은 Sensitive Code Split 과정으로 정적 오염분석 도구를 활용하여 데이터 유출의 위험이 있는 민감한 클래스와 메서드를 식별하고 애플리케이션 외부 서버로 코드를 분리한다. 두 번째 기법은 Screen Element Protection (SEP) 기법으로 Android UI를 보호할 수 있다. 세 번째 Android Data Storage Space Utilization (ADSSU) 기법은 안드로이드 데이터 공간을 활용해 데이터를 은닉할 수 있다. 마지막으로 Source Code Obfuscation 기법을 접목했을 때 자동화 분석 도구인 Android Monkey, UI Automator 같은 이벤트 기반 분석 도구들로부터 보안을 강화할 수

있다. 이를 통해 안드로이드 애플리케이션의 보안을 한층 더 강화하고 개인 정보 유출의 위험을 줄일 수 있다.

본 논문은 안드로이드 애플리케이션 보안 강화를 위해 강화된 난독화 기법을 소개하며, 이를 통해 최근 발생한 개인 정보 유출 사건과 같은 보안 위협에 효과적으로 대응할 수 있는 방안을 제시한다. 이 연구는 안드로이드 난독화 및 역난독화 기술의 지속적인 발전에 중요한 기여를 할 것으로 기대한다.

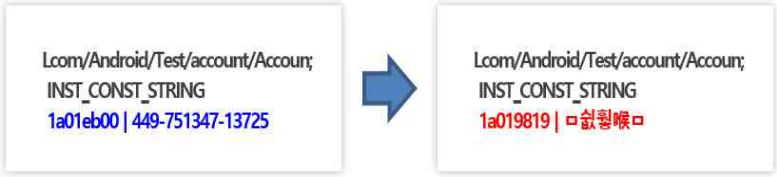
Ⅱ. 배경지식

1. 난독화 기법

1) 문자열 난독화

<그림 1>은 문자열 난독화 과정으로 “449-751…”와 같은 문자열이 알 수 없는 문자열(口섒훻喉口)로 변환되는 난독화 과정을 보여준다. 이러한 난독화가 적용된 문자열은 런타임 중 사용될 때 해독되어 사용된다. 문자열에는 비밀 키와 같은 중요한 정보가 포함되어 있으므로 이를 보호하기 위해 문자열 난독화가 빈번하게 사용된다. 또한 분석을 어렵게 하여 애플리케이션을 보호하거나 악성코드의 탐지를 회피하기 위해 사용된다(Dong et al. 2018; Bhakta et al. 2022).

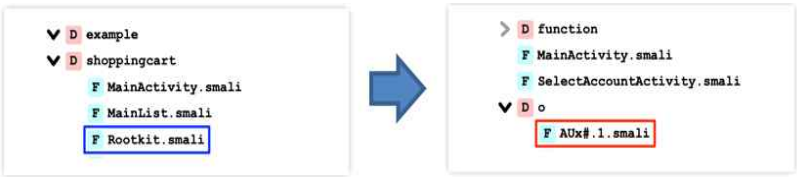
<그림 1> String Encryption



2) 클래스 난독화

<그림 2>는 클래스 난독화 과정으로 Rootkit.Class 파일을 식별하기 어려운 클래스(AUx#.1.class)로 변경하여 역난독화 공격으로부터 분석을 방지하는 난독화 기법이다. 클래스 난독화의 주된 목적은 코드의 이해와 분석을 어렵게 만들어 악의적인 목적으로 소스 코드를 사용하거나 분석하는 것을 방지한다(Faruki et al. 2014).

<그림 2> Class Encryption

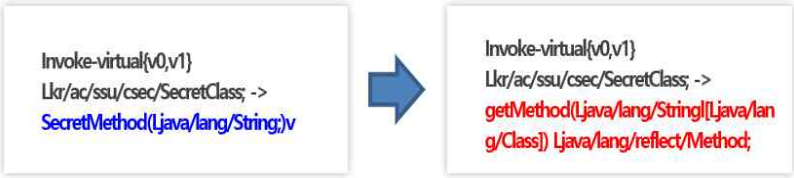


3) API 은닉

<그림 3>은 API 난독화 과정으로 SecretMethod를 GetMethod와 같이 변환하여 API 호출을 숨기는 기술이다. 일반적으로 Java Reflection과 JNI (Java Native Interface)를 사용하여 구현된다.

API 호출 및 모든 클래스의 멤버 또는 메서드에 대한 기타 접근을 숨기는 데 사용할 수 있다. 해당 기술을 적용할 경우 애플리케이션 공격자는 API를 호출하는 데 어려움이 존재한다(Anh et al. 2019).

<그림 3> API Hiding



2. 정적 오염분석

정적 오염분석은 소프트웨어 보안 분야에서 사용되는 기법으로, 애플리케이션에서 데이터가 어떻게 흘러가는지 추적하는 방법이다. 이러한 분석은 안전하지 않거나 신뢰할 수 없는 입력이 프로그램을 통해 어떻게 이동하는지 찾아내는 데 사용되며, 신뢰할 수 없는 데이터가 중요한 프로그램 영역으로 전파되지 않도록 하여 보안 취약점이나 잠재적인 공격 경로를 식별하는 데 도움을 준다.

안드로이드 애플리케이션에서 사용되는 정적 오염 분석 도구 FlowDroid(Arzt et al. 2014)는 Context, Control Flow, Fields, Objects에 민감하게 반응하여 애플리케이션 코드 내 가능한 오염된 흐름을 식별할 때 사용된다.

1) 소스와 싱크 분석

소스는 민감한 데이터가 시작되는 지점, 싱크는 그 데이터가 도달할 수 있는 민감한 지점을 의미한다. FlowDroid는 소스에서 싱크로 데이터가 어떻게 흐르는지를 추적한다(Li et al. 2015).

2) 콜백 분석

안드로이드 애플리케이션은 사용자 인터페이스 이벤트나 시스템 이벤트에 반응하기 위해 콜백 메서드를 사용한다(Huang et al. 2018). 콜백 분석은 이러한 콜백 메서드가 데이터 흐름에 어떻게 영향을 미치는지 조사한다.

3) 구성 요소 간 통신 분석

안드로이드 애플리케이션은 다양한 구성 요소(액티비티, 서비스, 송·수신 등)로 구성된다. 이러한 구성 요소 간의 통신은 Intent를 통해 이루어지며 정적 분석으로 해당 통신에서 발생할 수 있는 오염된 데이터 흐름을 추적하고 분석한다.

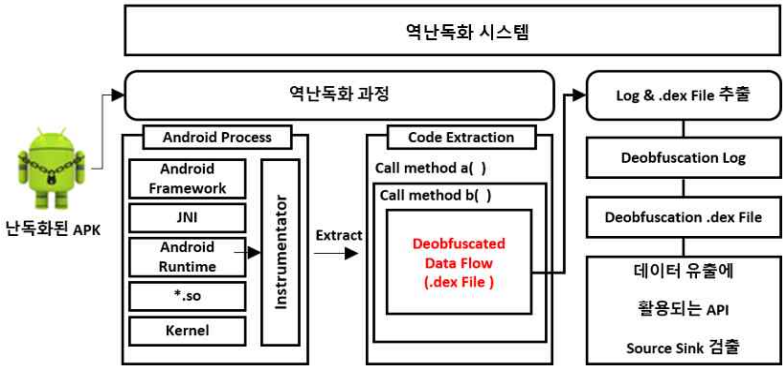
정적 분석을 통해, 구성 요소들이 서로 어떻게 주고받으며 애플리케이션의 다른 부분으로 데이터가 어떻게 흐르는지 분석할 수 있다.

3. 역난독화

<그림 4>는 기존에 존재하는 역난독화 도구로 동적 분석을 통해 사용된 문자열, 클래스, API를 분석하여 난독화된 애플리케이션을 다시 원본 형태로 복원하는 시스템이다(Lee et al. 2022).

역난독화는 주로 정·동적 분석, 그리고 디컴파일 등이 사용된다. 정적 분석은 애플리케이션을 실행하지 않고 코드를 분석하는 방법으로 디컴파일이나 디스어셈블과 같은 기술을 사용한다.

<그림 4> Deobfuscation System



정적 분석은 실행하지 않고 코드 분석이 가능하다는 장점이 있지만, 실행 중에만 나타나는 특정 동작이나 상태를 분석하기엔 어렵다는 단점이 있다(Li et al. 2017). 반대로 동적 분석은 애플리케이션을 실제로 실행하며 실행 중인 데이터를 모니터링하거나 조작하는 방법이다. 정적 분석으로 파악하기 어려운 정보를 얻을 수 있어 더 정밀한 분석이 가능하지만, 애플리케이션을 실행해야 하므로 분석 범위에 한계가 있다(Bierma et al. 2014).

최근에는 역난독화의 한계점을 극복하기 위해 자동화된 분석 도구들이 사용되고 있다. 다음의 세 섹션(2.3.1-2.3.3)에서 소개하듯이 자동화 분석 도구들을 이용한 역난독화 방법을 통해 난독화된 코드를 복원하는 것이 가능하다. 이에 따라 보안을 더욱 강화하고 강력한 난독화 기법을 개발해야 한다.

1) 정·동적 분석 도구

정·동적 분석 도구들은 난독화된 바이트코드를 역난독화하여 소스 코드로 변환한다. 정·동적 분석 도구는 디컴파일러와 역난독화 도구들로 분류된다. 디컴파일러는 JADX(jadx 2015), JD-GUI (JD-GUI 2019), JEB Decompiler(PNF Software 2013) 등이 존재한다. 역난독화 도구는 Tiro(David Lie, Michelle Y. and Wong 2018)와 Anti-Proguard(Baumann et al. 2017)가 존재한다.

Tiro는 난독화된 애플리케이션의 코드를 분석하고 변수, 클래스, Sensitive API 등을 분석하는 데 사용할 수 있다.

Anti-Proguard는 코드 최적화와 함께 식별자를 재정의하며 난독화된 코드를 분석하고 코드의 흐름을 쉽게 이해할 수 있다. 해당 도구들은 바이트코드를 원래의 자바 코드로 디컴파일을 시도할 수 있으며 애플리케이션 런타임에 디버깅도구를 사용하여 코드의 실행을 추적하여 난독화된 코드가 실제로 어떤 작업을 수행하는지 분석할 수 있다.

2) 난독화 식별 도구

난독화 도구들은 난독화 과정에서 특정 패턴이나 규칙을 사용하여 난독화를 진행한다. 난독화 식별 도구들은 이러한 패턴을 식별하여 어떤 난독화 도구가 사용되어 있는지 확인할 수 있다.

예를 들어, 난독화 식별 도구인 APKiD(APKiD 2016)를 이용했을 때 DexProtector(DexProtector Team 2011)와 LLVM-Obfuscation(Junod et al. 2015) 등 난독화가 적용된 경우 난독화가 적용되었음을 확인할 수 있다.

3) 자동화 분석 도구

자동화 분석 도구는 분석 과정을 자동화하여 원래 코드의 구조를 복원 및 분석할 수 있게 도와준다. 예를 들어 ProGuard, Dex Protector 등과 같은 난독화 도구가 적용되어 있는 애플리케이션을 자동으로 분석해 주는 도구로 UI Automator(Android Developers 2017), Android Monkey(Android Developers 2015) 등이 존재한다.

해당 자동화 분석 도구들은 난독화된 애플리케이션을 동적으로 실행시키면서 필요한 정보를 추출하고 역난독화를 진행한다. 따라서 자동화 분석 도구들은 역난독화 공격에 있어 필수로 방지해야 하는 기법 중 하나다.

4. 난독화 도구

Liapp(Liaap 2015)은 APK 파일을 입력으로 받는 난독화 도구로 소스코드를 보호하며 앱이 디컴파일 되지 않도록 보호한다. Liapp의 코드 보호는 클래스를 특정 위치에 숨겨 바이너리가 메모리에 로드되어 동적 분석기에서 민감한 데이터의 흐름에 대한 모니터링을 어렵게 한다.

Obfuscapk는 정적 난독화 도구로 코드 보안을 위해 CallIndirection, LibEncryption 등 다양한 기능을 제안하며, 패키지, 클래스, 메서드 이름 등을 식별자 변환하여 코드를 이해하는데 어렵게 한다.

DexProtector는 문자열 난독화, 클래스 난독화, API 은닉과 같은 코드 보호기능을 제공하여 분석에 있어서 분석가들로부터 분석을 어렵게 한다.

해당 기술들은 정적 분석은 물론이고 낮은 커버리지의 동적 분

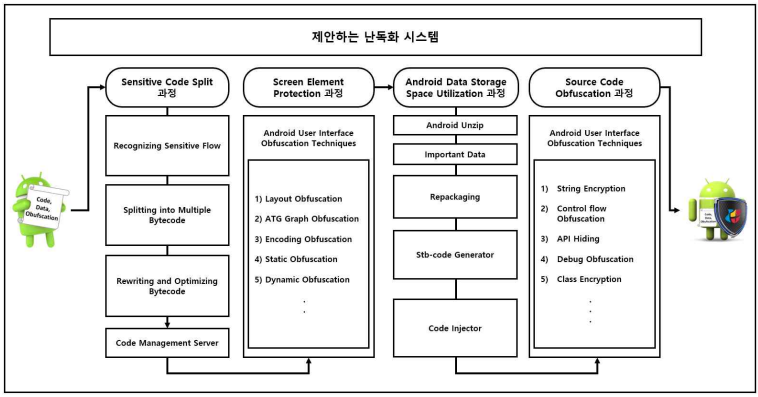
석까지 보호할 수 있다. 하지만 다양한 동적분석기(Obfuscation identifier Tool, JEB Decompilier, Deobfuscator 등)를 통해 분석할 수 있는 한계점이 존재한다.

따라서 제안기법은 동적 분석에 대한 분석률을 낮출 수 있도록 소스코드 분리 및 안드로이드 UI 인터페이스에 동적 암호화를 동시에 적용한다.

Ⅲ. 제안기법

앞서 기존의 난독화 방식이 역난독화의 위협에 대한 완전한 보안을 제공하지 못하는 것을 확인했다. 따라서 본 논문에서는 역난독화에 대한 저항력을 향상하기 위해 강화된 난독화 기법을 제안한다.

<그림 5> Proposed Obfuscation System

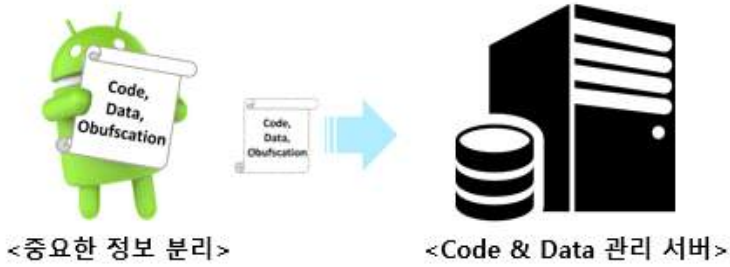


<그림 5>는 제안기법으로 네 가지 난독화 기법을 통합하여 기존 방법들보다 강화된 난독화 기능을 제공함과 동시에 각종 역난독화로부터 중요한 데이터 유출 및 각종 분석을 방지한다.

1. Sensitive Code Split 과정

민감한 코드의 분할은 데이터 보호에 있어 핵심적인 기능을 수행한다(Jeon et al. 2021). 이 기법은 FlowDroid를 활용하여 데이터 유출 가능성이 높은 중요 정보를 탐지하고 이를 보호하기 위해 Classes.dex 파일을 세 개의 별도의 .dex 파일로 분할한다.

<그림 6> Store Partitioned Important Data on the Server

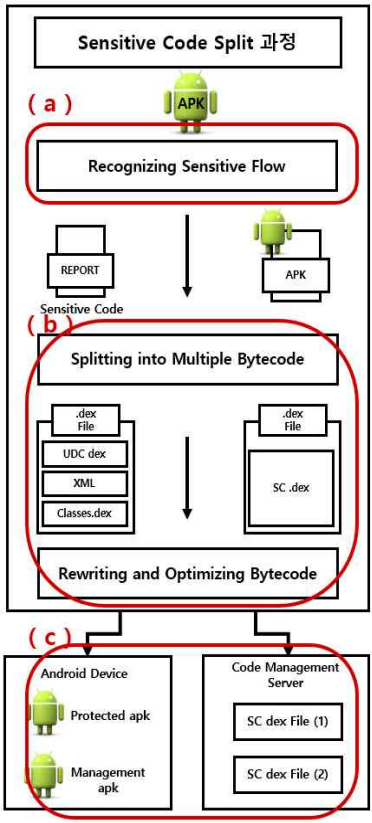


이 분할된 파일은 SC.dex (민감한 코드), UDC.dex (사용자 정의 클래스), 그리고 기본 Classes.dex로 구성된다. 사용자의 민감한 코드를 포함한 SC.dex 파일은 <그림 6>에 표시된 것처럼 안전한 코드 저장 서버에 보관된다. 서버에 SC.dex 가 저장되며 중요한 정보를 담은 분리된 코드를 실행시킬 수 있는 관리를 해주는 앱을 추가로 설치하게 된다.

<그림 7>은 Sensitive Code Split 과정을 설명한다. (a) 과정에

서는 FlowDroid에 APK를 삽입하면 Output으로 해당 APK와 Sensitive Code Report를 제공한다. (b) 과정에서는 APK에 존재하는 1개의 dex File을 3개의 dex File로 분리한다.

<그림 7> Sensitive Code Split Process



3개의 dex 파일은 SC.dex, UDC.dex, Classes.dex 파일로 구성된다. (c) 과정에서는 보호해야 할 SC.dex를 Code Managemen

t Server에 저장한다. Android Device에는 SC.dex가 분리된 APK와 코드를 관리하는 서버와 연동을 위한 Management APK가 디바이스에 설치된다. 민감한 코드를 분리한 후 애플리케이션의 해시값을 계산하여 무결성을 확인한다. SC.dex 파일과 해시값을 코드 관리 서버로 보내 저장하고 해시값을 쌍으로 관리한다. 코드 관리 서버는 Management APK로부터 SC.dex 파일에 대한 요청을 받으면 코드 관리 서버로부터 해시값을 계산하여 분리된 코드를 전송한다. 이를 통해 애플리케이션의 무결성을 확인한다.

Management APK는 민감한 코드를 수신한 후 Intent를 통해 분리된 코드를 전달한다. 결과적으로 애플리케이션이 원활한 동작을 위해, 분리된 코드가 필요할 때마다 Intent를 통해 Management APK에게 필요한 코드를 요청하고, Management APK는 코드 관리 서버로부터 필요한 분리된 코드를 요청하여 애플리케이션이 원활하게 실행될 수 있도록 한다.

이 과정은 난독화되거나 분리된 데이터를 코드가 관리되고 있는 서버에서 가져오는 절차를 포함한다. 서버에 보관되어 있는 데이터는 난독화되어 정·동적 분석을 통한 데이터 유출 위험으로부터 차단된다.

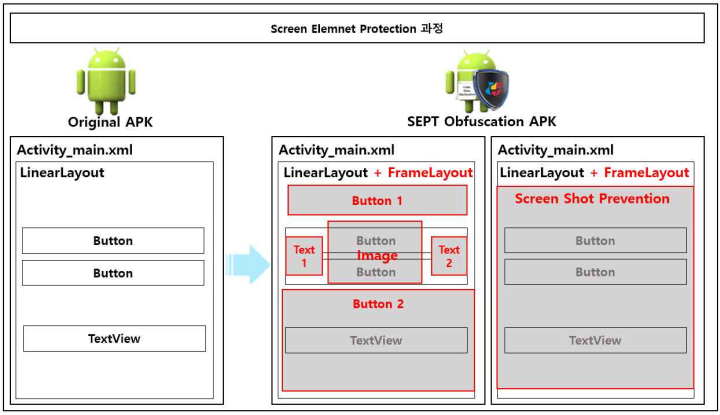
이러한 방법은 민감한 정보의 보호를 강화하고 안드로이드 애플리케이션의 보안을 강화할 수 있다. 그러나 Sensitive Code Split 과정의 한계점 또한 존재한다. Sensitive Code Split 기법은 자동화된 역난독화 또는 분석 도구들을 통해서 분석이 어려울 수 있지만, Frida 및 Android Open Source Project (AOSP)를 통해 분석하는 숙련된 역공학자에게는 분석이 가능할 수 있다.

2. Screen Element Protection 과정

Screen Element Protection (SEP) 과정은 안드로이드 인터페이스 요소들을 안전하게 보호하는 난독화 기법으로 <그림 8>과

같다(Hao et al. 2022). Android Monkey, UI Automator와 같은 역난독화 도구들이 수행하는 분석으로부터 안드로이드 사용자 인터페이스를 보호하는 것을 목적으로 한다.

<그림 8> Screeen Element Protection Process



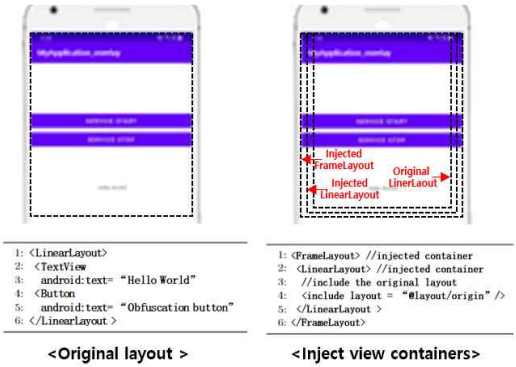
이 방법은 정·동적 분석 과정을 방해함으로써 보안을 강화한다. SEP 난독화는 소스 코드 난독화를 대체하는 방법으로 안드로이드가 UI 환경에 특화된 점을 이용하여 난독화를 진행한다. 이는 인터페이스 요소들을 의도적으로 복잡하게 만들어 공격자나 분석 도구들이 분석하기 어렵게 하고 자동화된 도구들이 흔히 사용하는 분석 기법의 취약점을 이용해 보안을 강화한다.

<그림 9>는 Linear Layout을 은닉하는 방법으로 중복 view container를 삽입함으로써 원본 Layout 파일에 view container를 추가하여 정적 view 계층 구조를 다르게 하고 런타임 시 view 계층 구조를 다르게 한다.

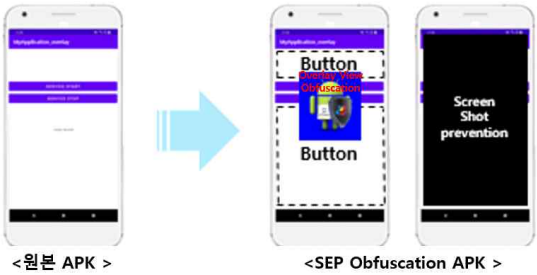
<그림 10>은 Activity Tree Graph 복잡도 증가, view 계층 복

잡화, Overlay view를 통한 UI 정보 은닉, 스크린샷 방지 기법을 적용한 그림으로 난독화가 적용된 경우 정·동적 분석 및 자동화된 동적 분석으로부터 분석을 방지할 수 있다.

<그림 9> Linear Layout Hiding



<그림 10> Before · After Applying SEP Obfuscation



Overlay view는 Services를 통해 최상단 위치에 view를 띄워 정보를 숨길 수 있으며, 스크린샷 방지 기법을 통해 자동화된 동적 분석을 방지할 수 있다. 따라서 SEP 난독화를 통해 개인정보 보호, 지적 재산 보호 등을 포함해 정보를 숨기거나 접근을 제한

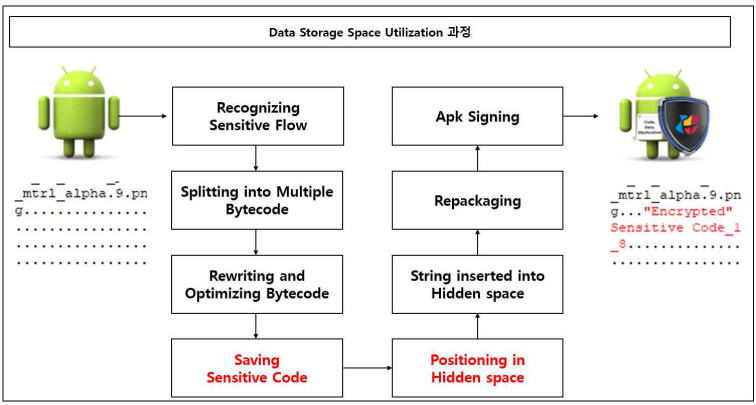
하는 데 활용할 수 있다.

이 과정에서 사용자의 사용에 방해되지 않도록 유의해야 하므로 사용자 인터페이스의 사용성이나 애플리케이션 성능에 부담을 주지 않는 선에서 SEP 난독화를 적용해야 한다.

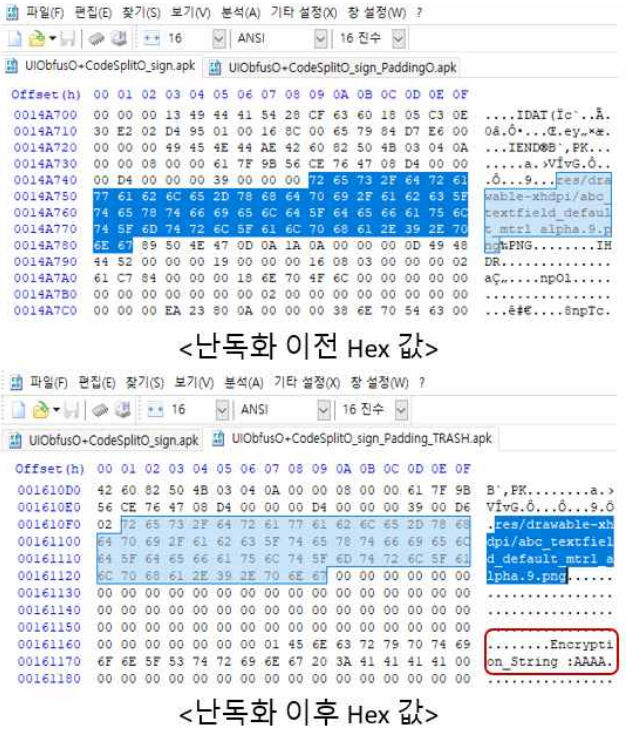
3. Android Data Storage Space Utilization 과정

<그림 11>은 Android Data Storage Space Utilization (ADSSU) 과정으로 스테가노그래피 기술을 사용한다. 해당 데이터는 안드로이드 내부 은밀한 곳에 저장하여 사용한 데이터 저장 공간을 활용하는 과정이다. 이 기법은 애플리케이션 내에서의 데이터와 자원들을 다른 파일이나 애플리케이션 내부에 숨기는 난독화 기술이다.

<그림 11> Data Storage Space Utilization Process



<그림 12> Before · After Applying ADSSU Obfuscation



ADSSU는 데이터를 “mtrl_alpha 9.png”와 같은 이미지 파일 뒤에 위치를 선정한 후, 해당 데이터를 이진 코드로 변환하는 작업으로 시작한다. 이 변환 과정을 거친 데이터는 인코딩되어 저장소에 배치되며 사용자 인터페이스 요소에 적용되기 위해 저장 공간 값들을 조정한다. ADSSU 난독화는 단순한 코드 난독화를 넘어서 애플리케이션 파일 구조의 데이터 저장 영역을 이용해 정보를 은닉하며 바이트코드와 같이 분석하기 쉬운 애플리케이션 코드의 보안을 강화하는 데 사용된다. 은닉된 데이터에 접근하기 위해서는

'Encrypted'라는 식별자를 남기고, 해당 식별자에 대응하는 값을 식별하여 숨겨진 정보를 검색할 수 있다.

이 과정은 중요한 데이터를 안전하게 저장하면서도 필요할 때 손쉽게 추출할 수 있는 유연한 방법이다. <그림 12>는 ADSSU 난독화가 적용되기 전과 후의 Hex 값 결과를 보여준다.

4. Source Code Obfuscation 과정

Source Code Obfuscation 과정은 ProGuard나 DexProtector와 같은 도구를 활용해 소스코드의 가독성을 낮추는 기법이다. 해당 과정을 통해 Android 애플리케이션의 코드를 이해하거나 수정하기 어렵게 만들어 무단 복제, 역난독화를 방지하는 과정이다.

<그림 13>는 제안기법에서 사용되는 문자열 난독화 알고리즘으로 String encoding을 활용하여 문자열 암호화(String Encryption) 난독화 기법을 적용했다. <그림 14>은 문자열 난독화 전·후 결과를 보여준다.

<그림 13> Encode String Constant

```
1: public void onclick (View view){
2:  Intent intent = new Intent();
3:  String encode =  "String Encryption Test" ;
4:  String decode = new StringBuilder(encode).reverse().toString();
5:  if(condition_1)
6:     intent.setClssName(S.getPackageName().decode);
7:  if(condition_2){
8:     Class<?>targetActivityClass= Class.forName(decode);
9:     intent = new Intent(S.this.targetActivitryClass);
10:   }
11:   startActivity(intent);
12:  }
```

<그림 14> Before · After String Obfuscation

```
# direct methds
.method static constructor <clinit>()V
    .registoers 3

    .prologue
    .line 15
    new-instance v0, Lcom/example/shoppingcart,

    const-string v1, "Activit_name"
```

<문자열 난독화 전>

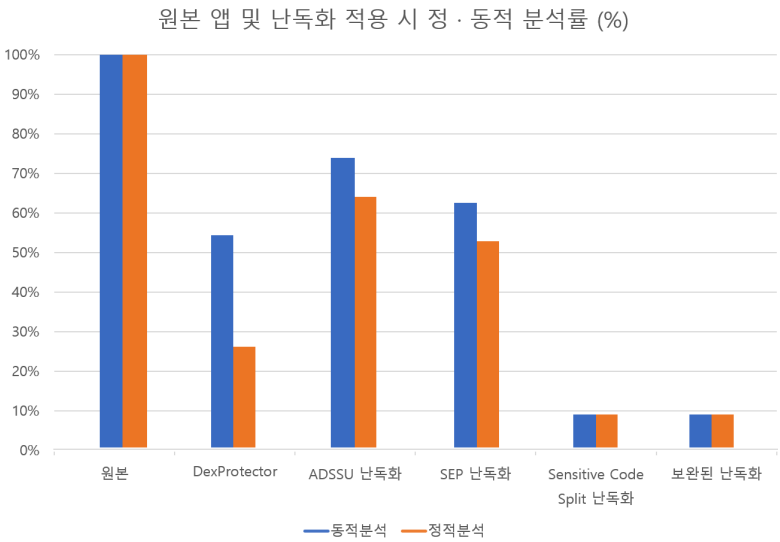
```
.method static constructor <clinit>()V
    .registoers 3

    .prologue
    .line 15
    new-instance v0, Lcom/example/shoppingcart,

    const-string v1, "eman_tivitca"
```

<문자열 난독화 후>

Ⅳ. 성능평가

본 섹션에서는 안드로이드 애플리케이션의 보안을 강화하기 위해 개발된 네 가지 기술을 접목한 강화된 난독화 기법의 효과를 확인한다. 사용된 환경으로는 Windows 10 pro, 12th gen intel (R) Core(TM) i9-12900 2.40GHz, 64GB, x64 기반 프로세스를 사용하였으며 모바일 기기는 Galaxy S9 SM-G960 모델로 삼성 엑시노스 9 Series (9810) SOC. 4GB LPDDR4X SDRAM, 64GB, 안드로이드 버전 10.0 버전으로 진행한다.

<그림 15> Analysis Rate When Obfuscating Is Applied



<그림 15>는 난독화가 적용되어 있지 않은 원본 APK, DexProtector의 Class Encryption 기법이 적용된 APK, ADSSU 난독화가 적용된 APK, SEP 난독화가 적용된 APK, Sensitive Code Split가 적용된 APK, 제안기법이 적용된 APK의 총 6개 APK를 대상으로 정·동적 분석률을 보여준다.

분석률은 원본 APK를 Android Studio를 이용하여 MainActivity까지 수동으로 실행했을 시 기록되는 로그를 기준으로 하여, 이와 동일하게 난독화된 APK의 로그를 비교하여 기록되는 로그의 양을 퍼센티지로 계산하였다.

원본의 APK의 경우 MainActivity까지 100%의 정보를 분석할 수 있었으며, Class Encryption이 적용된 DexProtector의 경우 난독화 기법이 적용되어 있어 MainActivity를 분석하기까지 정적 분석으로는 28%, 동적 분석으로는 55% 분석할 수 있다. ADSSU

난독화, SEP 난독화의 경우 앞서 설명한 섹션 3.2, 3.3과 같이 Control Flow를 복잡하게 하기 때문에 정적 분석은 각각 62%, 53% 분석률을 보이며, 동적 분석은 각 74%, 63%의 분석률을 보인다.

Sensitive Code Split 난독화가 적용되어 분석률을 측정한 결과 정적 분석과 동적 분석률은 각 8%를 기록하였다. Sensitive Code Split 난독화가 낮은 분석률을 보이는 이유는 SC.dex가 존재하지 않아 분석이 거의 불가능하다. 따라서 정상적으로 Sensitive Code Split이 적용된 APK를 실행하기 위해서는 사용자의 동의 후 실행할 수 있다. 최종적으로 제안하는 4개 기술이 접목된 난독화는 타 난독화 도구보다 현저히 낮은 분석률을 보인다.

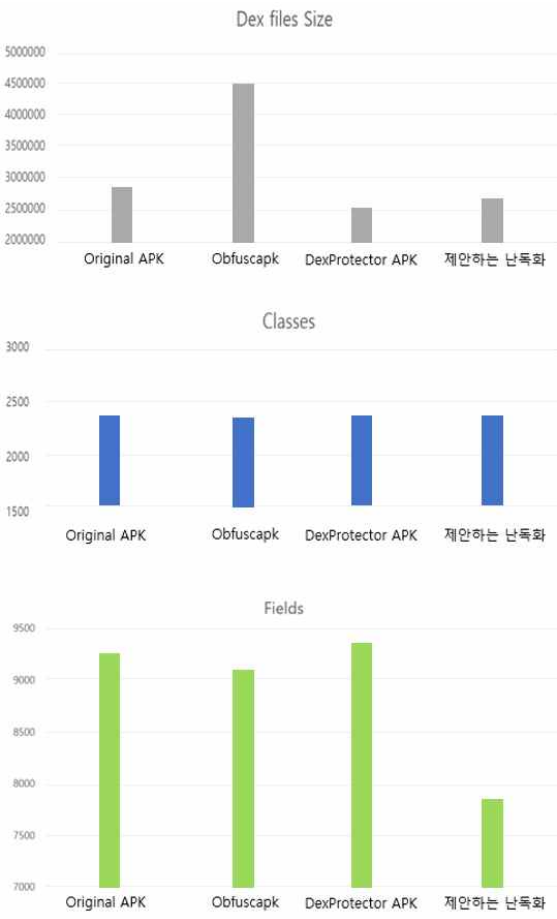
타 난독화 도구가 적용된 APK, 원본 APK, 제안하는 난독화를 적용한 APK들의 Classes.dex파일에서 인식할 수 있는 식별자를 비교한다. 이를 위해 Obfuscapk의 Reflection, DexProtector의 Class Encryption을 사용한다. 추가로 정적 분석 도구인 AndroGuard를 사용하여 제안하는 난독화 및 타 난독화 도구로 보호되는 Classes.dex파일에서 식별자(Dex File Size, Classes, Fields)를 추출하여 각 식별자의 개수를 보여준다.

<그림 16>은 타 난독화 도구가 적용된 APK들과 제안기법이 적용된 APK로부터 추출한 식별자들의 개수를 비교하였다.

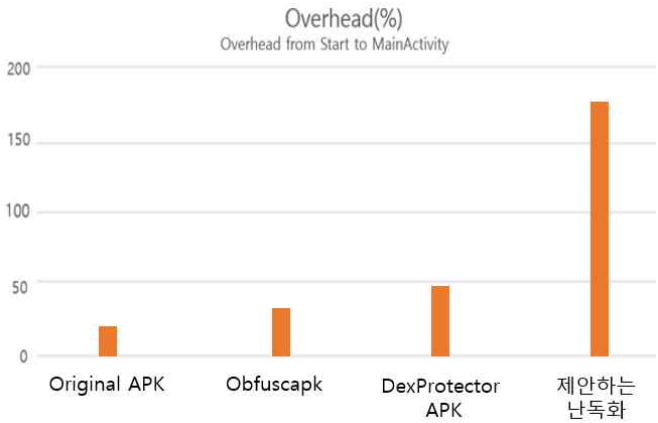
제안하는 난독화가 적용되어 보안이 강화된 APK의 Classes.dex 파일은 격리된 클래스로 인해 <그림 16>에 표시된 것처럼 클래스 및 필드를 포함하여 인식할 수 있는 식별자 수가 가장 적은 것으로 확인되었다. 이는 제안하는 난독화 기법이 적용된 APK가 정적 분석을 효과적으로 방지할 수 있다는 것을 보여준다. 이러한 분석률은 자동화된 분석 도구의 성능을 저하하고 역난독화 과정의 복잡성을 증가시키는 것으로 확인할 수 있다. 제안한 난독화 기법의 적용은 단순히 코드를 복잡하게 만드는 것을 넘어 애플리케이

션을 각종 역난독화 공격으로부터 효과적으로 보호한다.

<그림 16> Comparative Analysis of DEX File Metrics: Size, Classes, and Fields in Four APKs



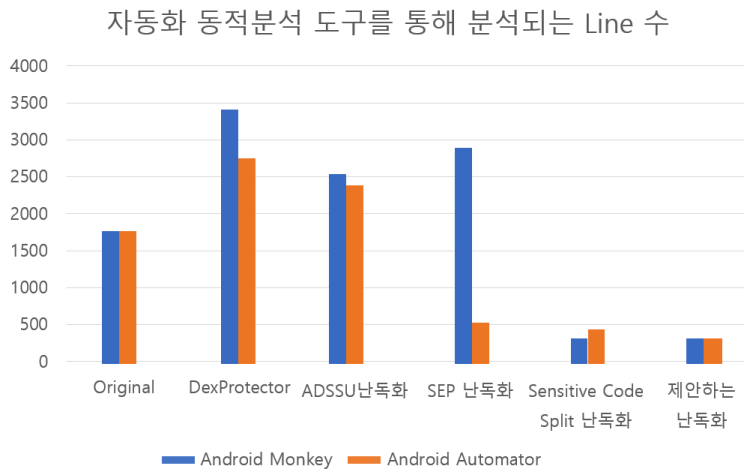
<그림 17> Measure Runtime for MainActivities in 4 Separate APK Files



<그림 17>은 각 난독화 도구에서 Class Encryption을 적용한 Test Sample 10개씩 사용하여 Overhead에 대한 성능을 평가했다. 성능 측정 방법으로는 Android Studio에서 발생하는 로그를 기반으로 MainActivity가 실행될 때까지 로딩 시간을 측정한다. 제안기법이 적용된 APK의 평균 로딩 시간이 오래 걸린 이유는 UDC.dex 및 SC.dex를 동적으로 로드하기 위해 추가적인 프로세스가 필요하며 코드 관리 애플리케이션에서 SC.dex를 요청 및 수신하기 위한 통신 프로세스가 필요하기 때문이다.

자동화된 분석 도구를 이용한 평가 방법으로는 현재 상용되고 있는 난독화 기법과 제안하는 난독화 기법이 적용된 APK를 사용하여 진행한다. 자동화 동적 분석 도구(Android Monkey, Android UI Automator)들을 이용하여 APK를 대상으로 10초 동안 자동으로 분석되는 로그를 측정하였다. 해당 과정에서 중복되는 로그는 제외하였다.

<그림 18> Evaluating Code Line Volume in Automated Dynamic Analysis



결과는 <그림 18>과 같이 측정되었다. Original APK의 경우 두 가지 자동화 분석 도구에 의해 분석했을 때 1500 line을 보였으며 DexProtector가 적용된 APK는 Control Flow가 복잡하게 난독화 되어 있어 2800 line 이상 로그를 보였다. 대표적으로 Sensitive Code Split 난독화가 적용되었을 경우 자동화 동적 분석 도구를 통해 분석이 거의 불가능하였는데 서버에서 분리된 코드를 가져와서 분석을 실행해야 하기 때문이다. 하지만 자동화 분석 도구에서는 서버에서 코드를 가져오는 기능을 지원하지 않기 때문에 분석이 불가능하다.

UI Automator를 이용한 분석에서 SEP 난독화의 분석률이 떨어지는 것은 스크린샷 방지 기능이 존재하기 때문에 그 결과 화면 스크린샷을 이용하는 UI Automator가 낮은 분석률을 보인다. 따라서 4개의 난독화 기술이 접목된 제안기법은 런타임 동작 중 Ov

erhead가 높다는 한계점을 가진다. 그러나 이를 제외하면, 기존의 난독화 방법들에 비해 정·동적 분석 및 역난독화에 취약하게 만드는 자동화 분석 도구들로부터의 보안을 강화하는 데에 있어서 상당한 이점을 제공한다는 것을 실증적으로 증명한다. 본 연구는 안드로이드 애플리케이션 보안성 향상을 위해 제안하는 난독화 기법의 유효성을 입증함으로써 보안 기술 발전의 향상을 기대한다.

Ⅴ. 결론

본 연구는 안드로이드 애플리케이션의 보안 강화를 위해 네 가지 기술을 접목한 강화된 난독화 기법의 유효성을 검증하고 결과를 분석하였다. 제안된 기법들은 단순히 애플리케이션의 중요한 부분들을 보호하는 것을 넘어 정·동적 및 자동화된 동적 분석을 통한 역난독화에 대한 저항력을 향상시켰다. 저항력의 향상은 기존의 난독화 방법들과 비교하여 눈에 띄는 보안 강화 효과를 나타냄으로 안드로이드 애플리케이션의 보안성을 한 단계 더 발전시키는 중요한 기여를 한다. 또한, 제안하는 난독화 기법이 실제 애플리케이션에 적용될 경우 사용자 데이터 보호와 애플리케이션의 전반적인 보안성을 강화하는 데 중요한 역할을 할 것으로 기대된다.

〈참고문헌〉

김영명. 2024. “ 애플리케이션 해킹 및 개인정보 유출 사건.” 『보안뉴스』. https://www.boannews.com/media/view.asp?idx=125433(검색일: 2024.1.31.)

Android Developers. 2015. "Monkey: Tool for Testing User Interface." https://developer.android.com/studio/test/monkey?hl=ko (검색일: 2023. 7. 26.)

Android Developers. 2017. "UI Automator for Testing User Interfaces." https://developer.android.com/training/testing/ui-automator?hl=ko (검색일: 2023. 7. 29.)

Anh, Pham, et al. 2019. “HideMyApp: Hiding the Presence of Sensitive Apps on Android.” *28th USENIX Security Symposium (USENIX Security 19)*.

Aonzo, Simone, et al. 2020. "Obfuscapk: An Open-Source Black-Box Obfuscation Tool for Android Apps." *SoftwareX* 11: 100403.

APKiD. 2016. "APKiD: Android Application Identifier for Packers, Protectors, Obfuscators and Oddities." https://github.com/rednaga/APKiD (검색일: 2023. 9. 10.)

Arzt, Steven, et al. 2014. “Flowdroid: Precise context, flow, field, object-sensitive and Lifecycle-Aware Taint Analysis for Android Apps.” *ACM Sigplan Notices* 49(6): 259-269.

Baumann, Richard, et al. 2017. “Anti-Proguard: Towards

Automated Deobfuscation of Android Apps.” In Proceedings of the *4th Workshop on Security in Highly Connected IT Systems*.

Bhakta, Dip, et al. 2022. "Android Malware Detection Agai nst String Encryption Based Obfuscation." *Congr ess on Intelligent Systems*, 543-555. Singapore: Springer Nature Singapore.

Bierma, Michael, et al. 2014. “Andlantis: Large-Scale And roid Dynamic Analysis.” *arXiv*:1410.7751.

DexProtector Team. 2011. "DexProtector - Mobile Applic ation Security." https://dexprotector.com/ (검색 일: 2023. 9. 3.)

Dong, Shuaike, et al. 2018. “Understanding Android Obfus cation Techniques: A Large-Scale Investigation in the Wild.” Security and Privacy in Communic ation Networks: 14th International Conference, 172-192.

Faruki, Parvez, et al. 2014. "Android Security: A Survey of Issues, Malware Penetration, and Defenses." *IEEE Communications Surveys & Tutorials* 17(2): 998-1022.

Hao, Zhou, et al. 2020. "UI Obfuscation and Its Effects on Automated UI Analysis for Android Apps." In Pr oceedings of the 35th IEEE/ACM International C onference on Automated Software Engineering, 199-210.

Huang, Huaxun, et al. 2018. “Understanding and Detecting

Callback Compatibility Issues for Android Applications." *ACM/IEEE International Conference on Automated Software Engineering*.

jadx. 2013. "jadx: Dex to Java Decompiler." https://github.com/skylot/jadx (검색일: 2023. 9. 12.)

JD-GUI. 2015. "JD-GUI: A Standalone Java Decompiler GUI." https://github.com/java-decompiler/jd-gui (검색일: 2023. 9. 11.)

Jeon, Geochang, et al. 2021. "Automated Multi-Layered Bytecode Generation for Preventing Sensitive Information Leaks From Android Applications." *IEEE Access* 9: 119578-119590.

Junod, Pascal, et al. 2015. "Obfuscator-LLVM--Software Protection for the Masses." IEEE/ACM 1st International Workshop on Software Protection. IEEE, 3-9.

Lee, Dongho, et al. 2022. "Deobfuscating Mobile Malware for Identifying Concealed Behaviors." *Computers, Materials & Continua*, 72(3).

Li, Li, et al. 2015. "Iccta: Detecting Inter-Component Privacy Leaks in Android Apps." In Proceedings of the *IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1.

Li, Li, et al. 2017. "Static Analysis of Android Apps: A Systematic Literature Review." *Information and Software Technology* 88: 67-95.

Liapp. 2015. "Liapp." https://liapp.lockincomp.com (검색일

2023.11.30.)
PNF Software. 2013. "JEB: Interactive Decompiler and Disassembler." https://www.pnfsoftware.com/jeb/demo (검색일: 2023. 9. 11.)
Wong, Michelle Y., and David Lie. 2018. “Tackling Runtime-Based Obfuscation in Android with TIRO.” *27th USENIX Security Symposium (USENIX Security 18)*.

논문접수일 : 2023년 11월 16일
논문심사일 : 2024년 1월 8일
게재확정일 : 2024년 1월 30일

Enhancing Android Application Security through Advanced Obfuscation Techniques

| Dongho Lee Soongsil University
| Haehyun Cho Soongsil University
| Kiwook Shon Seoul National University of Science and Technology

| Abstract

The rapid increase in mobile device usage has led to security threats, including the leakage of personal information. The reverse engineering of Android applications is becoming more sophisticated as it is used by attackers for malicious activities, posing a serious threat to cybersecurity. To counter these cyber threats, enhanced obfuscation techniques are needed to protect Android applications. This paper conducts empirical verification on how much the resistance against static and dynamic reverse engineering techniques is improved by applying four advanced obfuscation techniques. The proposed system includes Sensitive Code Obfuscation, Screen Element Protection, Android Data Storage Space Utilization, and Source Code Obfuscation, providing defense strategies against automated dynamic analysis tools and reverse engineering tools. The research findings reveal that the proposed obfuscation techniques considerably reduce the rate of analysis compared to traditional methods, offering significant improvements in securing against automated analysis tools. Such outcomes are anticipated to substantially contribute to the enhancement of Android application security.

| Keyword Reverse Engineering, Android, Obfuscation, Deobfuscation, Static Analysis, Dynamic Analysis, Static taint Analysis

* The authors extend their appreciation to all the colleagues who contributed to this paper and to the reviewers and editors who have assisted in its improvement. The authors wish to express their appreciation to the reviewers for their helpful suggestions which greatly improved the presentation of this paper