Android 앱 난독화 실험 및 분석 과제

보안개론 2분반, 32170578, 김산

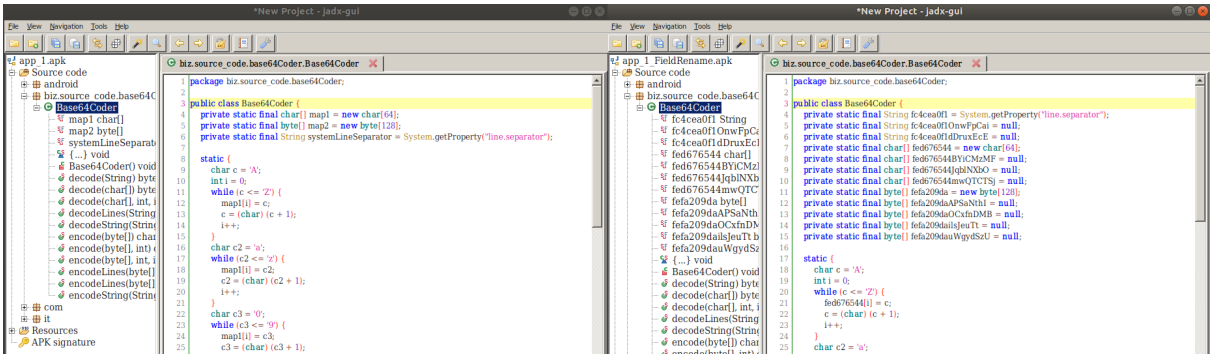
Obfuscapk 난독화 실험

1. Renaming

소프트웨어 개발단계에서 코드의 가독성을 위해 class, field, method와 같은 식별자는 사용용도를 쉽게 추측할 수 있는 의미있는 이름을 사용하게 됩니다. 이러한 의미있는 식별자 이름을 통해 쉽게 코드의 구조를 파악할 수 있고, 이용하려는 함수를 빠르게 찾을 수 있습니다.[1]

Renaming난독화 기법은 식별자의 이름을 기존 사용 용도와 무관한 이름으로 변경하여 소프트웨어에 대한 역공학을 어렵게하는 난독화 기법입니다.

A. FieldRename



[그림 1] jadx를 통한 apk파일 비교(좌: FeildRename 적용 전, 우: 적용 후)

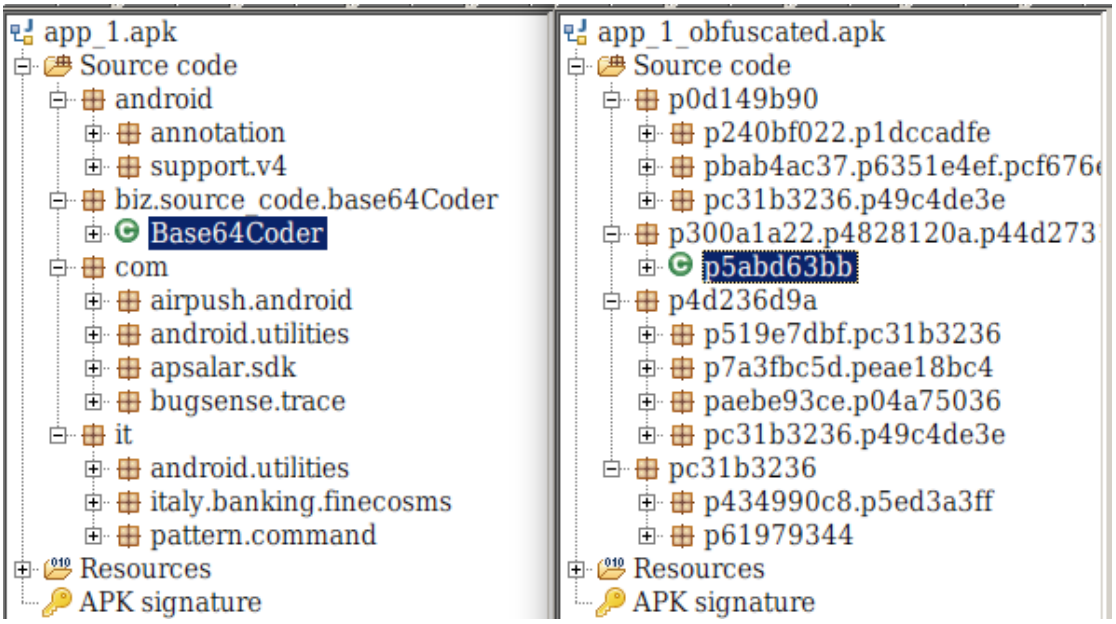
그림1에서 기존 파일에는 static 변수가 `map1`, `map2`, `systemLineSeparator` 3개만 있었지만

B. ClassRename

ClassRename의경우 `AndroidManifest.xml` 파일을 수정해야 하기 때문에 FieldRename에비해 좀더 복잡합니다.

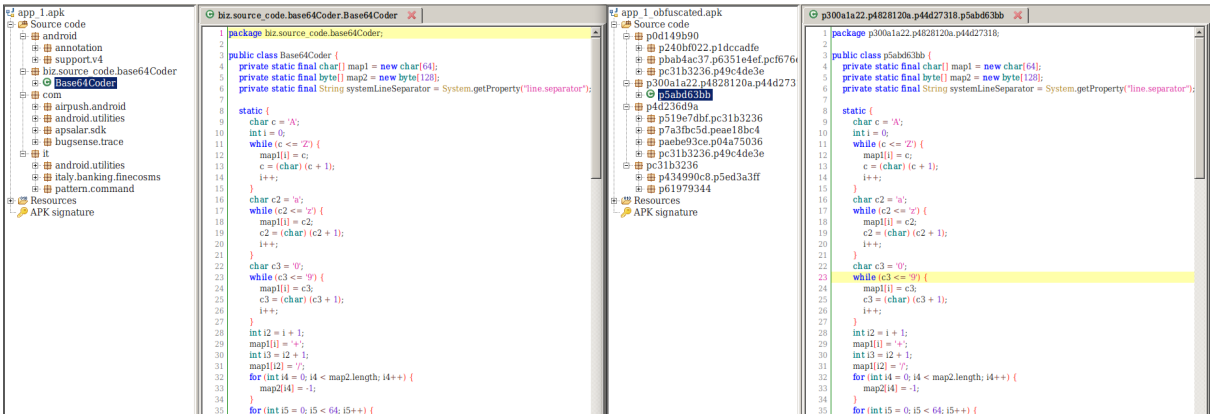
```
python3 -m obfuscapk.cli -o ClassRename ../../sample_apps/app_1.apk
```

`app_1.apk` 파일에 대하여 ClassRenaming난독화를 적용하였습니다.



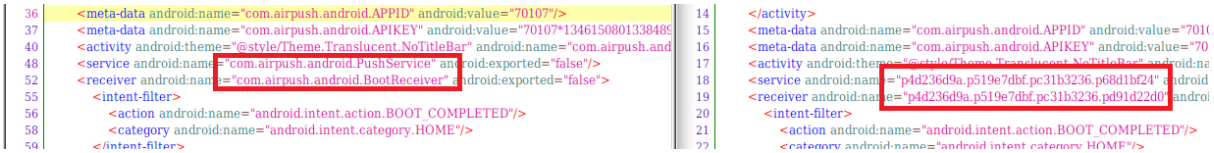
[그림 2] jadx를 통한 apk파일 비교(ClassRename, 좌: 적용 전, 우: 적용 후)

그림 1에서 난독화 전후를 비교했을때 패키지와 클래스의 개수가 동일합니다. Source code의 구조는 동일하지만 클래스의 이름이 사용용도와 무관한 이름으로 변경된 것을 확인할 수 있습니다.



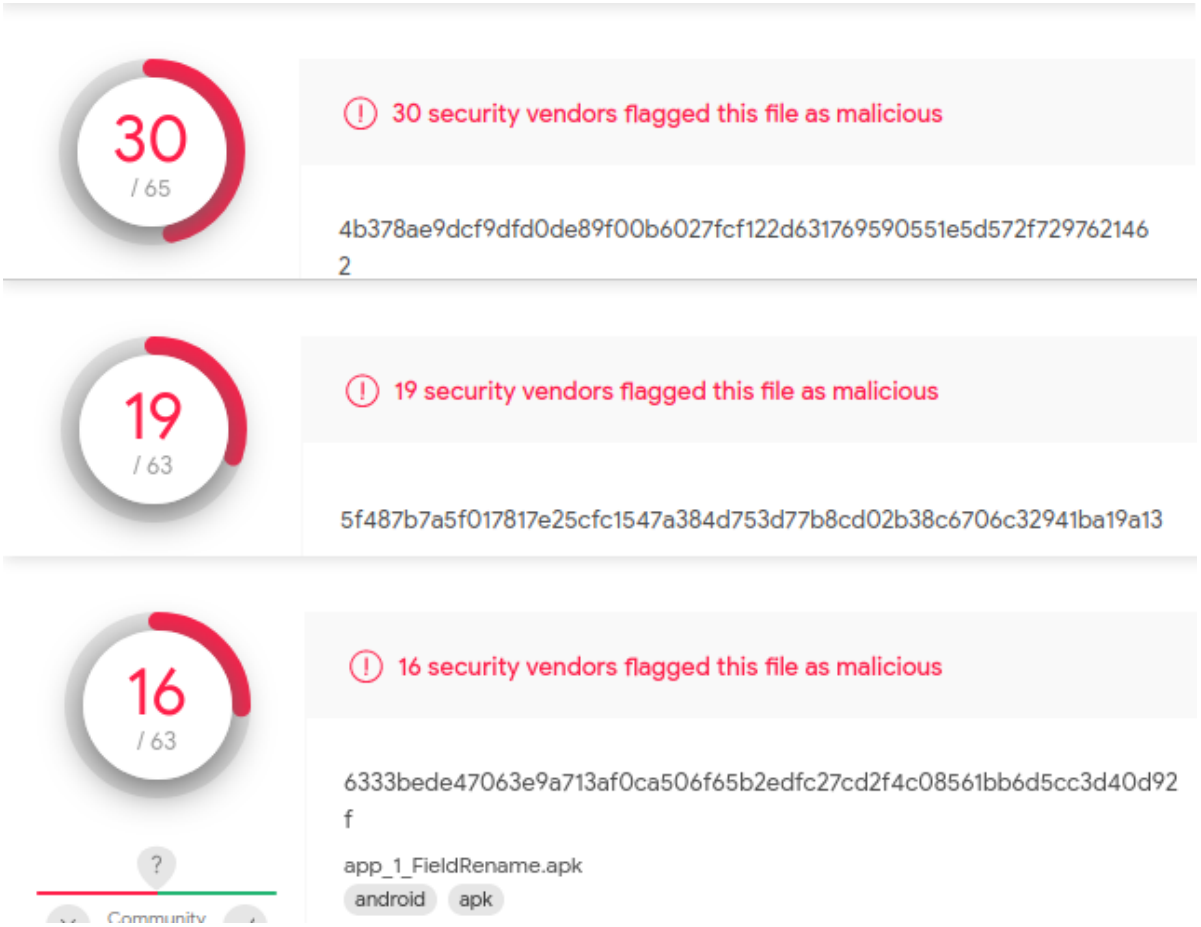
[그림 3] `Base64Coder`, `p5abd63bb` 클래스 내용 비교(ClassRename, 좌: 적용 전, 우: 적용 후)

그림3는 같은 파일이지만 이름만 다른것으로 추측되는 난독화 이전의 `Base64Coder`클래스와 `p5abd63bb`클래스의 내용을 비교한 그림입니다. 클래스 이름을 제외한 메소드, 변수 모두 동일한것을 확인할 수 있습니다.



[그림 4] AndroidManifest.xml파일 내용 비교(ClassRename, 좌: 적용 전, 우: 적용 후)

그림4에서는 `AndroidManifest.xml`파일의 클래스 이름이 난독화됨에 따라 바뀌어있는것을 보여줍니다.



[그림 5] Renaming 난독화 기법별 Virus total의 검사 결과 (위에서부터 적용전 , ClassRename, Field Rename 순)

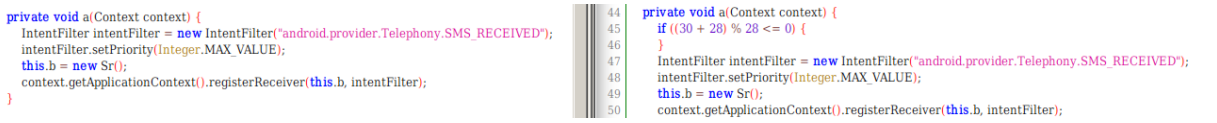
그림 5에서는 난독화 기법을 적용하기 전의 APK파일과 ClassRename을 적용한 파일, 그리고 FieldRename을 적용한 파일을 각각 VirusTotal에 업로드하여 악성 앱으로 판정하는 백신의 개수를 보여줍니다. 난독화를 적용한 후 백신이 악성앱으로 판정하는 비율이 낮아졌으며, ClassRename을 적용하였을 때 보다 FieldReaname을 적용하였을때 더 악성앱으로 판정되는 비율이 낮은것을 확인할 수 있었습니다.

2. Code

Obfuscapk의 code카테고리의 기법들은 classes.dex파일의 명령어에 영향을 주는 난독화 기법으로 code의 여러 부분에 적용되어 프로그램의 동작을 쉽게 파악할 수 없게합니다.

A. ArithmeticBranch

ArithmeticBranch기법은 실제로 분기하지 않는 분기문을 삽입하여 코드의 분석을 어렵게 하는 난독화 기법입니다.



```
private void a(Context context) {
    IntentFilter intentFilter = new IntentFilter("android.provider.Telephony.SMS_RECEIVED");
    intentFilter.setPriority(Integer.MAX_VALUE);
    this.b = new Sr();
    context.getApplicationContext().registerReceiver(this.b, intentFilter);
}
```

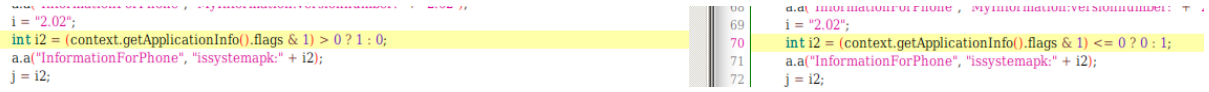
```
44  private void a(Context context) {
45      if ((30 + 28) % 28 <= 0) {
46      }
47      IntentFilter intentFilter = new IntentFilter("android.provider.Telephony.SMS_RECEIVED");
48      intentFilter.setPriority(Integer.MAX_VALUE);
49      this.b = new Sr();
50      context.getApplicationContext().registerReceiver(this.b, intentFilter);
```

[그림 6] jadx를 통한 apk파일 비교(ArithmeticBranch, 좌: 적용 전, 우: 적용 후)

그림 6에서 ArithmeticBranch를 적용한 우측 그림을 보면 기존에 없던 분기문이 추가된것을 확인할 수 있습니다. 추가된 분기문 `if((30 + 28) % 28 <= 0)`의 경우 내부의 산술 연산이 항상 거짓이 되어 어떠한 경우에도 분기하지 않는 더미 코드입니다. 이러한 더미코드의 삽입은 코드의 분석을 어렵게 하여 프로그램의 동작을 쉽게 파악할 수 없게 합니다.

B. Reorder

Reorder는 분기문의 조건을 바꾸는 난독화 기법입니다. 예를들어 기존 분기문이 `if(a < 0) ? 1 : 0`였다면 `if(a >= 0) ? 0 : 1`로 바꾸어 실행결과는 그대로 유지하면서 코드의 순서만 변경합니다.



```
i = "2.02";
int i2 = (context.getApplicationInfo().flags & 1) > 0 ? 1 : 0;
a.a("InformationForPhone", "issystemapk:" + i2);
j = i2;
```

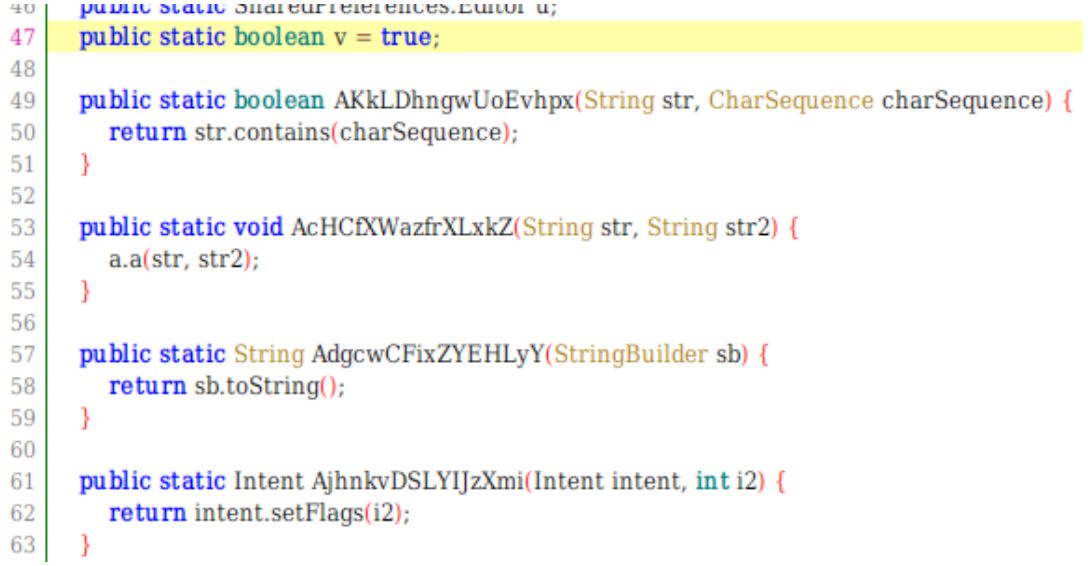
```
69  i = "2.02";
70  int i2 = (context.getApplicationInfo().flags & 1) <= 0 ? 0 : 1;
71  a.a("InformationForPhone", "issystemapk:" + i2);
72  j = i2;
```

[그림 7] jadx를 통한 apk파일 비교(Reorder, 좌: 적용 전, 우: 적용 후)

그림 7의 코드에서 기존의 코드는 `(context.getApplicationInfo().flags & 1) > 0 ? 1 : 0`으로 연산의 결과가 0보다 크다면 참에 해당하는 1을 반환하고 아니라면 거짓에 해당하는 0을 반환합니다. 변경된 코드는 `(context.getApplicationInfo().flags & 1) <= 0 ? 0 : 1`으로 연산의 결과가 0보다 작거나 같다면 참에 해당하는 0을 반환하고, 그렇지 않으면 거짓에 해당하는 1을 출력합니다. 이는 조건문은 다르지만 같은 결과를 출력합니다. 이와같이 Reorder는 조건문의 실행결과는 유지하면서 조건은 바꾸어 난독화하는 기법입니다.

C. CallIndirection

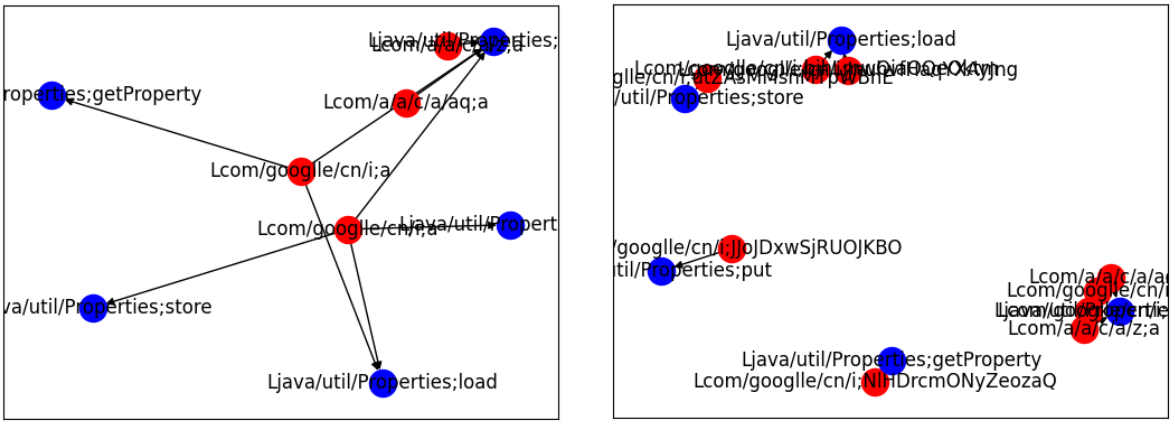
CallIndirection은 기존 코드의 역할을 유지하면서 제어흐름을 변경하는 난독화 기법입니다.
예를들어 기존에 m_1이라는 함수가 호출되어 실행되었다면 난독화 이후에는 m_2라는 wrapper함수를 따로 만들어 m_1을 호출하게 하여 m_1을 직접 호출하지 않고 m_2를 통해 m_1을 호출하게 됩니다.



```
47  public static boolean v = true;
48
49  public static boolean AKkLDhngwUoEvhpx(String str, CharSequence charSequence) {
50      return str.contains(charSequence);
51  }
52
53  public static void AcHCfXWazfrXLxkZ(String str, String str2) {
54      a.a(str, str2);
55  }
56
57  public static String AdgcwCFixZYEHLyY(StringBuilder sb) {
58      return sb.toString();
59  }
60
61  public static Intent AjhnkvDSLYIJzXmi(Intent intent, int i2) {
62      return intent.setFlags(i2);
63  }
```

[그림 8] CallIndirection난독화 후 jadx를 통한 apk파일 분석

그림 8은 Call Indirection을 기법을 적용한 후 기존의 메소드를 직접 호출하지 않고, 임의로 만든 Wrapper함수를 통해 메소드를 호출하는것을 보여줍니다.



[그림 9] androguard를 통한 control flow graph 비교(좌: 난독화 전, 우 : 난독화 후)

그림 9는 androguard의 `Create Call Graph from APK` 기능을 이용하여 `app_2.apk`의 `Ljava/util/Properties` 클래스의 Call graph를 보여주고 있습니다. 난독화 이전에는 제어흐름을 쉽게 파악할 수 있지만, 난독화 이후에는 wrapper함수로 인해 제어흐름을 파악하기 어렵습니다.

D. MethodOverload

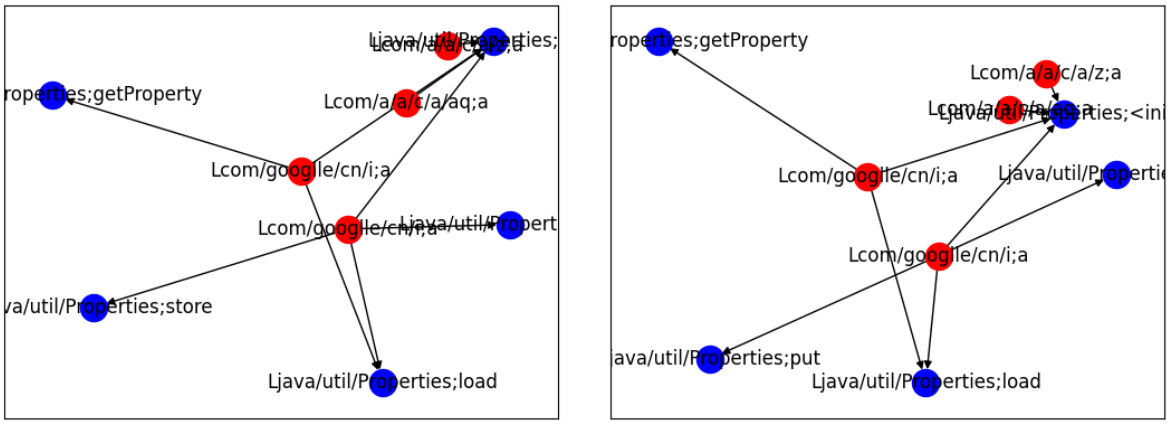
MethodOverload는 java의 다형성을 활용한 기법입니다. 기존함수와 동일한 함수 이름에 인자만 다른 함수를 정의하여 실행흐름을 쉽게 파악하기 힘들도록 만듭니다. [2]

```
public static String f() {
    try {
        Enumeration<NetworkInterface> networkInterfaces = NetworkInterface.getNetworkInterfaces();
        while (networkInterfaces.hasMoreElements()) {
            Enumeration<InetAddress> inetAddresses = networkInterfaces.nextElement().getInetAddresses();
            while (true) {
                if (inetAddresses.hasMoreElements()) {
                    InetAddress nextElement = inetAddresses.nextElement();
                    if (!nextElement.isLoopbackAddress()) {
                        c.ca.b.a.a("InformationForPhone", "□□□ip□□□" + nextElement.getHostAddress().toString());
                        return nextElement.getHostAddress().toString();
                    }
                }
            }
        }
    } catch (SocketException e) {
        c.ca.b.a.a("InformationForPhone", "□□ip□□□□□□□□□□□□");
        e.printStackTrace();
    }
    return null;
}

public static void f(char c2, int i, boolean z, byte b2) {
    double d = (double) ((42 * 210) + 210);
}
```

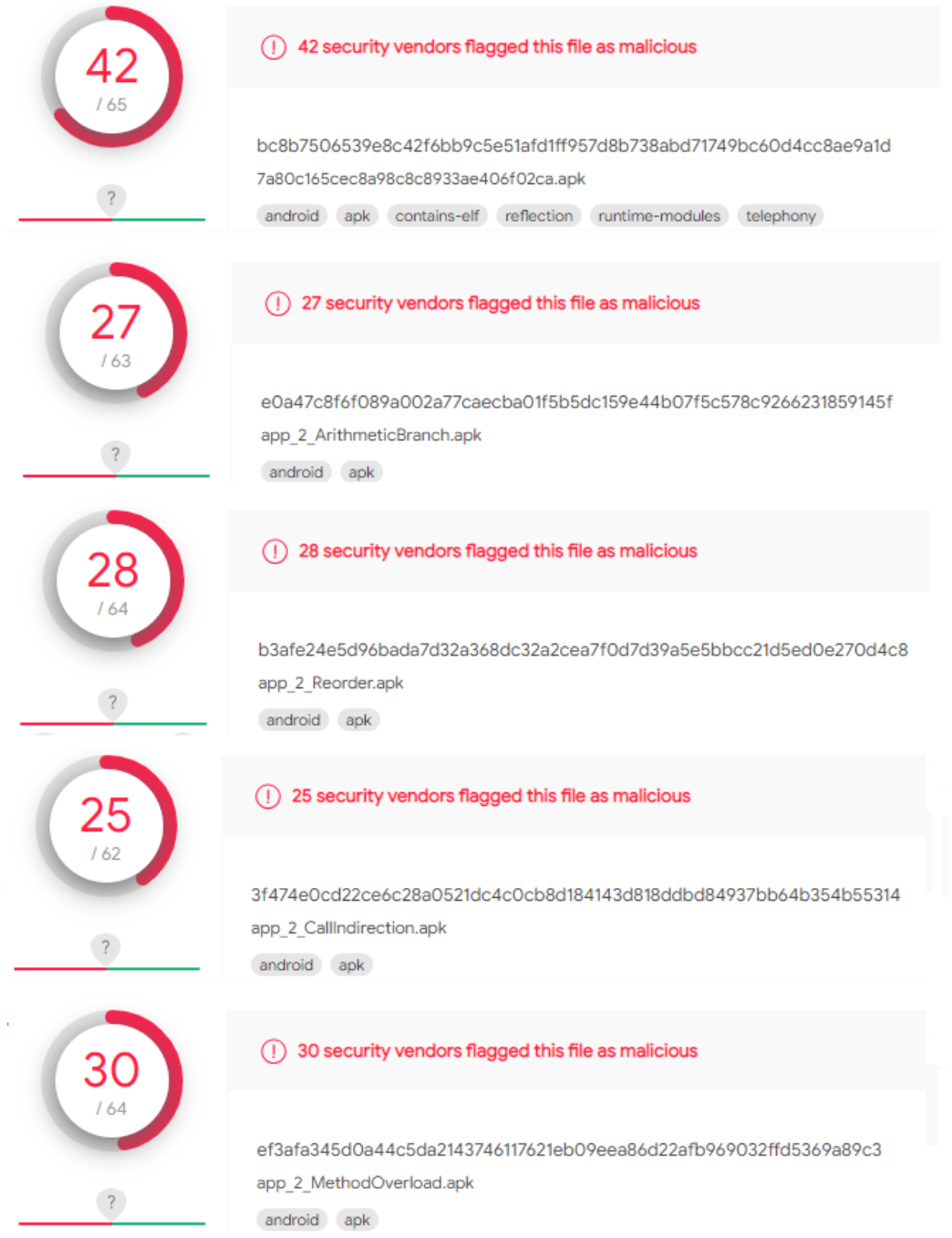
[그림 10] MethodOverload 난독화후 같은 이름을 가진 오버로딩 함수가 생성된 모습

그림 10은 MethodOverload난독화 후 기존의 `f()` 메소드와 동일한 이름에 인자수가 다른 `f(char, int, boolean, byte)` 함수가 아래에 생성된 모습입니다. 아래의 새로 생성된 오버로딩 메소드의 경우 랜덤한 산술 연산 이 함수 내부에 존재하지만 실제 실행흐름에는 아무 영향을 끼치지 않습니다.



[그림 11] androguard를 통한 control flow graph 비교(좌: 난독화 전, 우 : 난독화 후)

그림 11은 androguard의 `Create Call Graph from APK` 기능을 이용하여 `Ljava/util/Properties` 클래스의 Call graph를 보여줍니다. 난독화 전,후 실행 흐름이 바뀌지 않은것을 확인할 수 있습니다.



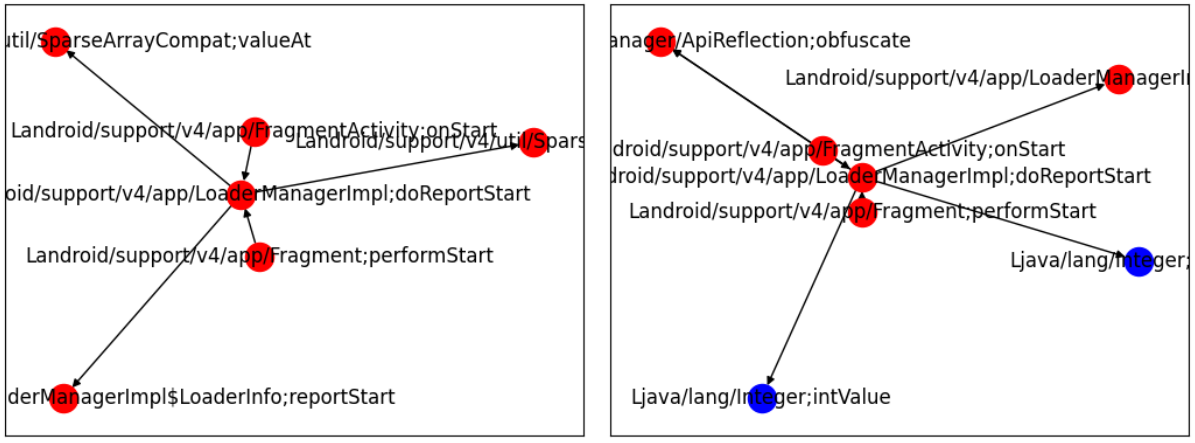
[그림 12] Code 난독화 기법별 Virus total의 검사 결과 (위에서 부터 적용전 , Reorder, ArithmeticBranch,Reorder, CallIndirection, MethodOverload 순)

3. Reflection

Reflection

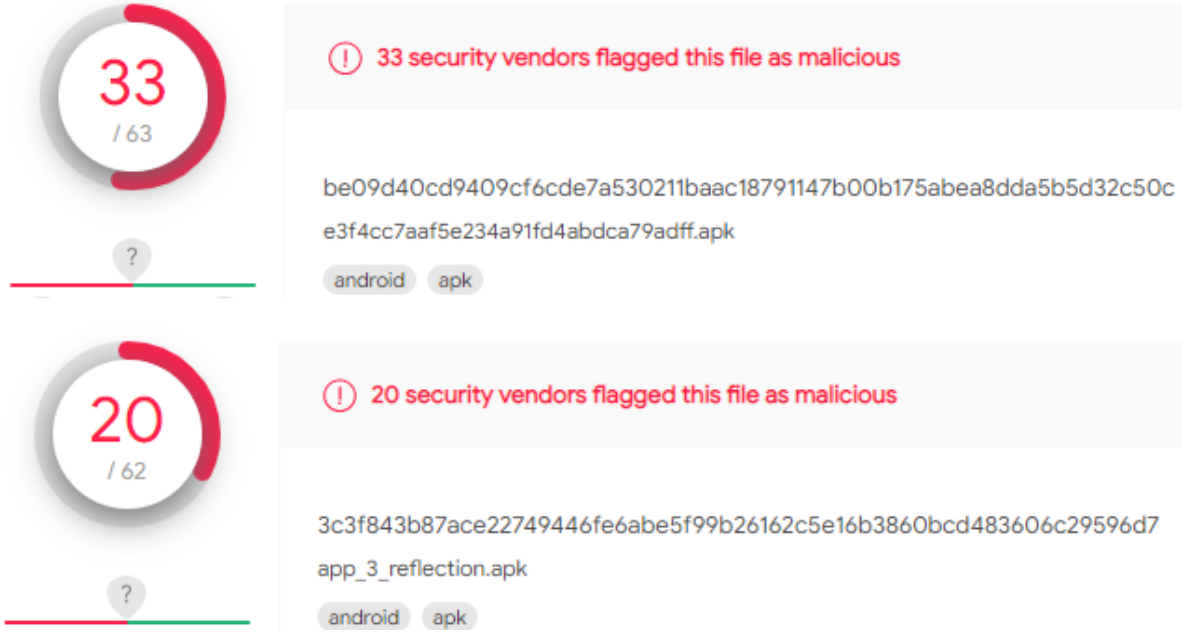
Reflection은 자바 프로그래밍 언어의 특징으로 런타임에서 자바 프로그램의 객체나, 메소드와 같은 정보들을 파악할 수 있게 해줍니다. 예를들어 메소드의 정보를 알고싶을 때 `.getName()`, `.getDeclaringClass()`와 같이 메소드의 이름이 무엇이고 어떤 `class`에 선언되어 있는지 확인할 수 있습니다.

이러한 자바의 reflection을 이용하면 보다 많은 호출 단계를 거치는 구조로 변형되기 때문에 난독화 효과를 얻을 수 있습니다.[3] Obfuscapk에서는 reflection api를 통해 난독화가 이루어집니다.



[그림 13] androguard를 통한 control flow graph 비교(reflection, 좌: 적용 전, 우: 적용 후)

그림 13은 `doReportStart()`메소드의 제어흐름을 비교한 그림으로, `ApiReflection.obfuscate()`를 통해 난독화가 이루어지는것을 확인할 수 있습니다.



[그림 14] reflect 난독화 기법의 Virus total의 검사 결과 (적용전, 적용후)

난독화 실험 및 분석 과제를 통해 느낀 점

개발자 관점

이번 과제에서 제공된 5개의 애플리케이션 파일을 `jadx`를 통해 열어서 코드의 내용을 확인할 수 있었는데, 그만큼 쉽게 다른 공격자들이 정상적인 애플리케이션에 악성 코드를 삽입하여 악용할수 있다는 생각이 들었습니다. 특히 이러한 자바를 통해 개발된 애플리케이션의 경우 디컴파일러를 사용하면 제가 과제를 하면서 보았던 것 처럼 거의 소스코드 그대로 나오기 때문에 악용이 더 쉬울것 같습니다.

개발자의 관점에서 본인이 개발한 애플리케이션이 공격자들에 의해 악용되어 악성앱으로 배포가 되는 것을 막기 위해서는 코드 난독화를 통해 공격자들이 애플리케이션의 쉽게 동작방식을 이해하여 악성코드를 삽입하지 못하도록 해야합니다. 다만 reflect와 같은 기법을 통해 난독화를 하게 된다면, 물론 보안성이 그만큼 좋아지지만, 추가적인 함수 호출로 인한 오버헤드가 발생하기 때문에 적절하게 보안성을 고려하여 난독화를 하는것이 좋을 것 같습니다.

또한 공격으로부터의 방어뿐만 아니라 코드에 포함되어 있는 지적재산권이 유출되지 않아야 한다면 이러한 난독화 기술을 유용하게 사용할수 있어 보입니다.

만약 백신 개발자라면 이러한 난독화를 통해 의도를 숨긴 악성 소프트웨어를 감지하기 위해 난독화된 코드를 분석할 수 있는 알고리즘을 구현해야 할 것입니다. 따라서 java만의 reflect난독화 처럼 각 언어별 특성에 대해 잘 이해해야할 뿐만 아니라 최근의 난독화 트렌드에 대해서도 잘 파악하고 있어야 할 것입니다.

공격자 관점

공격자는 악성 앱을 배포하면서 백신에 의해 쉽게 차단되지 않도록 애플리케이션의 코드를 난독화할 수 있습니다[4],[5]. 난독화 기법에 따라 악성 소프트웨어의 의도를 감지하기 어렵게 하여 백신의 탐지시간을 지연시키고, 탐지를 회피할 수 있기 때문입니다.

또한 난독화되어 배포된 애플리케이션을 잘 분석하여 공격하기 위해서는 이러한 난독화 코드를 잘 이해할 수 있도록 공격자는 난독화와 관련된 개발 트렌드를 잘 이해하는것이 중요할 것 같습니다.

개인적인 생각으로는 reflect난독화를 심하게 하여 백신의 동적 분석 시간을 지연시키고, 그동안 추가적인 공격을 하는 방법이 좋을것 같다는 생각이 들었습니다. 취약점이 백신으로 인해 방어가 가능하다면, 시스템이 백신에 할당할 수 있는 자원은 시스템에 따라 한정적이기 때문에 백신은 이러한 성능 오버헤드를 고려하여 실시간 검사를 수행할 것입니다. 때문에 동시에 검사할 수 있는 악성 응용프로그램에 제한이 있기 때문에 이러한 취약점을 잘 활용한다면 좋을것 같습니다.

참고문헌(Reference)

[1] Aonzo, S., Georgiu, G. C., Verderame, L., & Merlo, A. (2020). Obfuscapk: An open-source black-box obfuscation tool for Android apps. SoftwareX, 11, 100403. doi:10.1016/j.softx.2020.100403

[2] Chua, M., & Balachandran, V. (2018). Effectiveness of Android Obfuscation on Evading Anti-malware. Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy - CODASPY '18. doi:10.1145/3176258.3176942

[3] Lee, Joohyuk, & Park, Heewan. (2015). 리플렉션과 문자열 암호화를 이용한 안드로이드 API 난독화 도구. 정보처리학회논문지:컴퓨터 및 통신 시스템, 4(1), 23–30. https://doi.org/10.3745/KTCCS.2015.4.1.23

[4] M. Schiffman, "A Brief History of Malware Obfuscation: Part 2 of 2," http://blogs.cisco.com/security, Feb. 2010.

[5] W. Wong and M. Stamp, "Hunting for Metamorphic Engines," Journal in Computer Virology, vol. 2, no. 3, pp. 211-229, Dec. 2006.