Universitat Autònoma de Barcelona

Dipòsit digital de documents de la UAB

This is the **published version** of the bachelor thesis:

Marín Gual, Àlex; Ventayol Marimón, Jordi, dir. Obfuscador Java. 2023. (958 Enginyeria Informàtica)

This version is available at https://ddd.uab.cat/record/272798

under the terms of the BY-NC-ND license

OBFUSCADOR JAVA/KOTLIN

Àlex Marín Gual

Resum– Avui en dia, els smartphones formen part de la vida de tothom. Aquests dispositius els tenim plens d'aplicacions que poden tenir informació personal que és vital que es mantingui privada. Per això, és important assegurar un nivell de seguretat per a les aplicacions. La manera més comuna amb la que buscar vulnerabilitats en una aplicació és a través de l'enginyeria inversa, un procés on s'intenta deduir el funcionament d'un sistema a través de l'observació i raonament. Si una persona aconsegueix el codi font d'una aplicació, podria analitzar el codi per trobar dades desprotegides que podrien trobar-se en clar o buscar vulnerabilitats. Una manera molt eficaç per a evitar aquests atacs és en la que està centrat aquest TFG, la obfuscació de codi. Amb l'obfuscació de codi podem aplicar modificacions al codi font per crear la màxima confusió possible a l'hora d'intentar entendre què fa el codi, dificultant considerablement els atacs d'agents maliciosos.

Paraules clau– Obfuscació, Aplicacions Android, Java, Kotlin, Seguretat de Codi, Enginyeria Inversa

Abstract– Nowadays, smartphones are part of everyone's life. These devices are full of Apps that can have personal information which is vital that remains private. Therefore, it is important to ensure a level of security for Mobile Apps. The most common way to search for vulnerabilities in an application is using reverse engineering, a process where one tries to deduce the behaviour of a system through observation and reasoning. If a person gets the source code of an App, they could scan the code to find unprotected data or look for vulnerabilities. A very effective way to avoid these attacks is using code obfuscation, the method explained in this report. With code obfuscation, we can apply modifications to the source code to create as much confusion as possible when trying to understand how the compiled code behaves, considerably hindering attacks by malicious agents.

Keywords– Obfuscation, Android Apps, Java, Kotlin, Code Security, Reverse Engineering

1 INTRODUCCIÓ

LES aplicacions Android que tenim en els nostres telèfons mòbils són instal·lades utilitzant el que s'anomena APK. Una APK (Android Package) és un conjunt de fitxers que contenen tots els documents necessaris per poder dur a terme la instal·lació correctament de l'aplicació, contenint aquests documents les classes, llibreries i metadata.

Una persona maliciosa pot fer ús de descompiladors per a tal d'aconseguir una versió de codi d'alt nivell d'una APK concreta. D'aquesta manera, es pot analitzar molt fàcilment el comportament de l'aplicació desitjada, permetent observar tant funcionalitats com mesures de seguretat empleades. Per tant, és molt important aplicar una capa de seguretat per no fer tan senzill aquest procés d'enginyeria inversa.

Aquí entren en joc els obfuscadors de codi. Aquests tenen la funcionalitat d'afegir canvis en el codi resultant a l'hora de compilar una aplicació. Aquestes alteracions tenen com a objectiu crear confusió i dificultar que una persona sigui capaç d'analitzar el funcionament del codi obtingut a través de descompilar la APK.

• E-mail de contacte: alex.maring@autonoma.cat
• Menció realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Jordi Ventayol Marimón
• Curs 2022/23

1.1 Estat de l'art

Hi ha diferents tècniques d'obfuscació, com per exemple *string obfuscation, control flow o function merging*, entre moltes altres. *String obfuscation* és el procés on s'apliquen modificacions als Strings que estan en clar per tal de que no es vegin què son a simple vista. *Control flow* consisteix en canviar l'estructuració del codi per tal de crear el que és conegut com a "espagueti", molt rebuscat i desestructurat, fent que no es vegi clar què fa cada cosa. Finalment, *function merging* consisteix en fusionar diferents funcions en una sola, creant una nova crida per a especificar quines funcions executar en cada cas. Aquest cas concret permet fer inútil l'anàlisi del gràfic de trucades a funcions, ja que només es cridarà a la mateix funció contínuament, sense deixar clar un ordre de jerarquia.

1.2 Objectius

L'objectiu final del TFG serà començar a desenvolupar un obfuscador aplicable sobre aplicacions Android de Java/Kotlin, centrant-se en la tècnica de *String obfuscation*. Amb aquest obfuscador s'aconseguirà afegir una capa de protecció davant atacs d'enginyeria inversa. Per això, s'analitzaran diferents obfuscadors públics amb aquesta tècnica amb la intenció d'avaluar el comportament de cadascun, especialment com manipulen el codi per a dur a terme l'obfuscació dels Strings i, finalment, escollir quin aporta més seguretat.

S'ha escollit aplicar *String obfuscation* i no un altre mètode perquè és el mètode d'obfuscació que aplica un canvi més important de cop. A més, alguns obfuscadors que apliquin tècniques diferents podrien interferir negativament entre ells, anul·lant els seus efectes mútuament.

1.3 Metodologia

Per aconseguir l'obfuscador, primer es necessitarà crear una aplicació senzilla [1] sobre la que treballar, per poder aplicar els tests i conèixer el seu funcionament. Aquesta consistirà en una aplicació de Registre i Log-in, on es poden crear usuaris amb contrasenya i, si s'introdueixen aquests paràmetres correctament en la funcionalitat d'inici de sessió, es valida el procés. Com que l'aplicació tracta amb informació sensible, com una contrasenya, es podrà veure la importància que tindrà protegir als usuaris de que un atacant conegui el procés de validació.

Per a començar a comprendre els efectes d'un obfuscador, es començarà comparant els resultats d'haver compilat l'aplicació amb i sense ProGuard, un obfuscador lliure que introdueix canvis senzills i és l'opció per defecte en el software d'Android Studio. Fent això, s'aconseguirà una noció dels efectes que té un obfuscador sobre el resultat de la compilació i les diferents parts de les que està formada la APK.

Després s'haurà de dur a terme algun cas d'enginyeria inversa. Com que a la universitat no hi ha cap assignatura que ensenyi en què es basa aquest procés, primer es farà un curs online gratuït per aprendre les nocions bàsiques i els conceptes principals. Amb aquests nous coneixements, es podrà començar a comparar les diferències entre el procés de descompilació sobre una APK obfuscada i una en clar.

Amb aquests resultats, es podrà seguir a obtenint més informació sobre els obfuscadors i quines alternatives hi ha disponibles. Finalment es podrà buscar els punts febles de cada obfuscador i es podrà decidir l'opció més adient per a utilitzar en les nostres aplicacions Android.

1.4 Organització del document

A la primera secció d'aquest informe s'explica les funcionalitats que s'han tingut en compte per a crear una aplicació per a poder fer tests posteriorment amb diferents obfuscadors. També s'explica quines classes componen l'aplicació per facilitar la comprensió a l'hora de dur a terme els anàlisis de resultats.

A la següent secció es fa una comparació dels resultats d'haver generat la APK amb la utilització del ProGuard i sense. Seguidament, es mostren les modificacions necessàries en una aplicació per tal d'habilitar l'ús d'un obfuscador de Strings en la compilació.

A la quarta secció s'explica detalladament el procediment que es segueix per a dur a terme un atac d'enginyeria inversa, juntament amb el software necessari per a realitzar-lo. Finalment, a l'última secció es fa un conjunt d'atacs reals d'enginyeria inversa sobre APKs creades a partir de diferents obfuscadors. D'aquests atacs s'extrauran les conclusions necessàries per a poder escollir un obfuscador adient.

2 CREACIÓ DE L'APLICACIÓ DE TEST

L'aplicació que s'utilitzarà per dur a terme les execucions futures del treball ha sigut creada en Android Studio amb Kotlin.

Com que no es tenia coneixement previ de com utilitzar ni Android Studio ni Kotlin, la creació de l'aplicació es farà partir de l'article proporcionat per la pàgina de desenvolupadors d'Android [2]. En aquest *codelab*, es comença a partir de la plantilla anomenada *Basic Activity* disponible en el software, on existeixen dos fragments inicials. En el primer hi ha un botó que et porta al segon fragment, on en aquest últim es pot llegir el text de *Hello world*.

Seguint el tutorial, juntament amb els coneixements obtinguts sobre llenguatges de programació, es podrà anar deduint com crear nous fragments, noves funcionalitats i fins i tot el tractament de variables a través d'*inputs* d'usuari.

2.1 Requisits funcionals

Les funcionalitats principals de l'aplicació són les de registre i inici de sessió d'usuaris, per tant, l'aplicació ha de comptar amb una 'pantalla' per al registre d'usuaris i una altra per a l'inici de sessió. Les 'pantalles' d'una aplicació Android s'anomenen fragments. També és necessari un fragment principal, que permetrà seleccionar quina de les dues funcionalitats es vol utilitzar.

Pel que refereix al registre d'usuaris, s'ha de permetre introduir un nom d'usuari i una contrasenya i notificar a l'usuari del resultat del procés de registre, és a dir, si s'ha pogut crear correctament l'usuari o no. També ha de permetre retornar al fragment principal si es decideix no registrar cap usuari.

D'altre banda, el fragment de log-in ha de permetre la introducció del nom d'usuari i contrasenya i verificar que coincideixen amb les dades d'un usuari ja existent. De la mateixa manera que el cas del fragment de registre, aquest fragment ha de permetre tornar al fragment principal si es decideix no utilitzar la funcionalitat d'inici de sessió.

A la figura 1 es poden veure aquests fragments i quines transicions entre ells es poden dur a terme.

2.2 Requisits no funcionals

Per a tal de tenir un registre d'usuaris, l'aplicació compta amb un fitxer inicial situat a la carpeta d'*assets* del projecte, amb text en format JSON per tal d'indicar els camps de nom d'usuari i contrasenya per a cada entrada. JSON és un format de text senzill amb l'objectiu de facilitar l'intercanvi de dades. Aquest fitxer serà utilitzat per a tenir usuaris

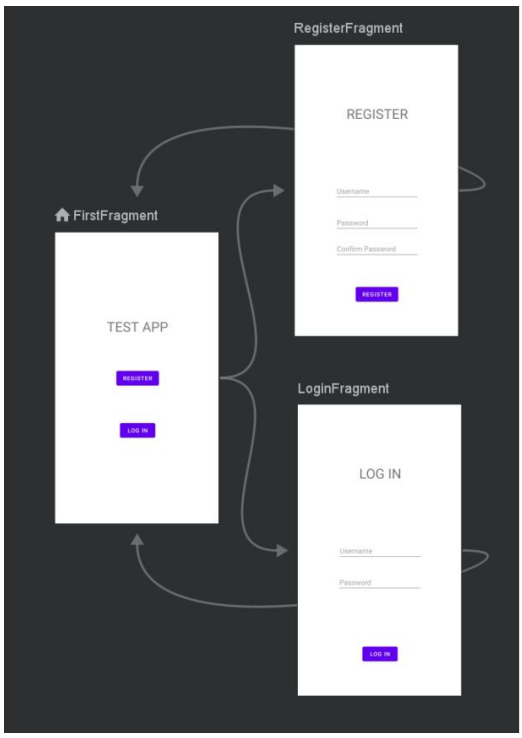

Fig. 1: Fragments de l'aplicació

creats automàticament la primera vegada que s'utilitzi l'aplicació en un nou dispositiu. S'utilitza la carpeta d'*assets* perquè Android Studio permet accedir als continguts d'aquest directori per a només lectura a través del context del projecte.

A l'hora de fer el registre d'usuari, s'ha de comprovar que les dues entrades introduïdes per a la contrasenya coincideixen i que el nom no ha sigut utilitzat prèviament i, en cas negatiu, comunicar a l'usuari per a quina raó no s'ha pogut dur a terme el registre.

Els usuaris són emmagatzemats en el mateix format JSON mencionat prèviament, a la capacitat de magatzem interna del dispositiu, dins d'un directori anomenat ACCOUNTS. D'aquesta manera, la localització del fitxer serà coneguda en cas de que el directori de memòria interna estigui massa desorganitzat. Cada vegada que es generi un nou usuari, s'haurà d'afegir al final d'aquest fitxer l'entrada corresponent del nou usuari.

Per a dur a terme l'inici de sessió, un cop s'hagi clicat el botó corresponent, s'agafen els valors introduïts per l'usuari i s'haurà de comprovar si aquests existeixen en el fitxer d'usuaris. En el cas que es compleixi, es notifica a l'usuari de manera positiva, i en el cas que un component no permeti la verificació de l'usuari, també s'ha de notificar.

2.3 Classes

L'aplicació compta amb un total de 5 classes. A la secció A.1.1 de l'apèndix es mostra el diagrama de classes per donar una idea més visual sobre el funcionament de les classes i com interactuen entre elles.

La classe mainActivity és igual a la del projecte per defecte. S'encarrega de crear la barra superior de l'aplicació per a tal de facilitar la navegació entre fragments.

La següent classe s'anomena FirstFragment. Aquesta classe re-aprofita la implementació proporcionada del disseny per defecte per a la gestió de transicions entre fragments. A més, se li ha afegit el procés de comprovar si existeix un fitxer en magatzem intern del dispositiu que controli els usuaris i les seves contrasenyes. Si no existeix, es crea inicialitzant-lo amb els valors creats per defecte. Aquesta classe també gestiona la funcionalitat dels dos botons, permetent anar a qualsevol dels altres fragments.

La classe encarregada de gestionar els registres d'usuari s'anomena RegisterFragment. Aquesta té implementada la funcionalitat del botó que, un cop clicat, adquireix els valors introduïts en els camps de text de la vista per a comprovar que el registre es pot dur a terme de manera correcte. Per començar, es llegeix el text del fitxer de comptes, es separa el JSON en els diferents usuaris i s'iteren tots. Si el nom d'usuari introduït coincideix amb algun ja existent, el registre falla. De la mateixa manera, si les dues contrasenyes no coincideixen entre elles, el procés també falla. Si cap de les dues condicions anteriors es compleix, es pot afegir al fitxer organitzador el nou usuari. Depenent de quin cas ha seguit l'execució, es mostrarà a l'usuari un missatge o un altre.

L'última classe que defineix un fragment és la LoginFragment. De manera semblant a la classe mencionada anteriorment, aquesta classe gestiona la funcionalitat del botó que, quan clicat, llegirà el fitxer de comptes, iterarà sobre tots els usuaris registrats i si en algun moment es troba amb un usuari que coincideixin els valors introduïts en text, notificarà que l'inici de sessió ha tingut èxit. De manera contrària, si l'usuari no existeix o la contrasenya ha sigut introduïda de manera incorrecta, també es notificarà de que l'inici no s'ha dut a terme. Per tal de no proporcionar informació extra, el missatge d'error serà genèric, per no permetre diferenciar quin cas incorrecte ha ocorregut.

Finalment, hi ha una classe anomenada JSONController, la qual és l'encarregada de fer totes les lectures i escriptures de fitxers. En aquesta classe hi ha la informació de tots els directoris necessaris per poder accedir als arxius i també té implementada tota la funcionalitat de JSON necessària. Les tres classes anteriors, quan necessiten fer una lectura o escriptura per a poder executar una funcionalitat, deleguen aquestes operacions a aquesta classe.

A l'apartat A.1.2 de l'apèndix es mostra l'aparença de l'aplicació i com interactuen els elements de cada vista.

3 ANÀLISIS DE COMPILACIÓ

Per a començar a fer l'anàlisi dels resultats de crear la APK, primer s'ha de compilar. Per tal d'especificar la configuració que haurà de tenir la compilació, es pot modificar l'arxiu de *build.gradle* [3]. Un paràmetre que es pot modificar en aquest arxiu és si es permet l'execució de les normes del ProGuard. ProGuard és un obfuscador de Java/Kotlin que està implementat per defecte en Android Studio. Es pot especificar si s'executarà, i quins arxius s'utilitzen per detallar les normes que s'hauran de seguir, i alguns paràmetres addicionals [4].

Per a tal d'obtenir una APK amb obfuscació i una altre sense, s'ha compilat dues vegades, una amb l'opció de *minifyEnabled* a *true* i l'altre a *false*. Quan el valor està definit com a *true*, el ProGuard està habilitat per a poder fer modificacions en el resultat de la compilació. Per poder analitzar ambdues APKs s'ha utilitzar l'analitzador proporcionat pel

propi Android Studio i un software de descompilació de Java anomenat JADX.

Un aspecte que s'ha de tenir en compte és que hi ha dues maneres de crear la APK: en mode *debug* i en mode *release*. El mode *debug* s'utilitza per a fer compilacions ràpides, degut a que normalment s'utilitza per aplicar canvis petits en el codi. Degut a això, aquest mode crea una APK que no aplica el ProGuard, ja que provocaria una compilació més lenta. Per tant, hem de compilar en mode *release* si volem veure l'efecte complet de l'obfuscador.

Les diferències més notables a simple vista són les que estan relacionades amb l'ocupació dels fitxers dins de les APKs. El pes de l'APK generada utilitzant ProGuard és de 2,7 MB, mentre que si l'obfuscador no ha sigut utilitzat, genera una APK de 5,2 MB, gairebé ocupa el doble.

Addicionalment, també es pot veure que quan s'utilitza ProGuard només es genera un sol arxiu *.dex* on es declaren i s'emmagatzemen les classes, i si no s'utilitza se'n generen 4. A més, les classes creades per a la funcionalitat de l'aplicació estan localitzades a classes3.dex, ja que classes.dex hi ha totes les classes de Google, Android i més generals.

3.1 Efecte del Minify

Per tal d'analitzar les APKs s'ha utilitzat l'eina anomenada JADX [5], una eina que converteix els arxius *.dex*, els quals tenen tots els recursos de les classes d'una aplicació, a codi Java. D'aquesta manera, es pot analitzar una versió semblant al codi utilitzat per a crear l'aplicació, tot i que no coincideixen completament la versió proporcionada per JADX i la utilitzada originalment.

Per començar, analitzarem les diferències que es poden apreciar a l'hora d'organitzar de classes. Com s'ha mencionat anteriorment, el directori de treball era *com/example/login_test*, per tant, si volem avaluar els resultats, hem d'analitzar aquest directori. A primera vista ja es pot observar una diferència important, i és que la versió obfuscada no té cap classe anomenada JSONController. L'operació de *minify* ha decidit prescindir d'aquesta classe i repartir les seves funcionalitats en altres classes.

A continuació, observarem la classe FirstFragment, concretament la seva funció on es produeix la crida de la classe JSONController. En la versió sense obfuscar, dins de la funció *onCreateView*, que s'executa cada vegada que s'executa aquest fragment, es crea un objecte de la classe JSONController per tal de poder comprovar si existeix el fitxer necessari per guardar usuaris. La funció de JSONController utilitzada s'anomena *initFiles*. Si mirem com es mostra aquesta funció a JADX, es veu clarament que fa una lectura d'un fitxer anomenat *accounts.json*, ja que es crea un objecte de tipus *BufferedReader*, i després s'escriu en un arxiu anomenat *accountFiles.txt*, ja que es crea un *FileInputStream*. A més, es pot observar que tant el nom de les variables com el de la funció no ha sigut alterat.

Si ara observem la mateixa funcionalitat en la versió obfuscada, observem unes quantes alteracions. Per començar, en la pròpia classe FirstFragment no es troba a simple vista la funció *onCreateView* degut a la obfuscació. En aquesta versió, la classe FirstFragment té les funcions constructora i unes altres funcions anomenades *C*, *t* i *u*, les quals reben paràmetres d'entrada diferents. Comparant amb la versió original, podem saber que la funció *onCreateView* rep per paràmetre un *LayoutInflater*, un *ViewGroup* i un *Bundle*, i retorna una *View*. Per tant, podem deduir que la funció *onCreateView* ha sigut rebatejada a *t*, ja que reben els mateixos paràmetres d'entrada.

D'altre banda, també s'observa que en cap moment es fa una crida a un constructor d'alguna classe sense cap paràmetre, que ens indicaria l'existència de la classe de JSONController. En la mateixa funció de FirstFragment es crea un *InputStream* cap a un fitxer anomenat *accounts.json*, per tant, podem assegurar que el *minify* ha considerat òptim implementar la funcionalitat de la classe directament en aquesta nova ubicació. Això es degut a que la funció *initFiles* es cridava sempre que es cridava a la classe FirstFragment, no era necessari en el *frontend* prémer cap botó per que s'executés la funcionalitat. Per tant és millor a l'hora de compilar que la funcionalitat estigui implementada directament en el mateix lloc que la creació de la vista del fragment. A més a més, de la mateixa manera que abans, els noms de les variables s'han conservat i els Strings no han rebut cap modificació en aquesta classe.

A continuació analitzarem el cas d'una altra funcionalitat que sí calgui la interacció de l'usuari per a tal d'ésser executada, la funció *checkPassword*, que es crida des de la classe LoginFragment.

Començant un altre cop per la versió sense obfuscar, en aquesta ocasió es fa la crida de la classe JSONController des de la funció *onViewCreated* de la classe LoginFragment. Com s'ha mencionat anteriorment, el resultat proporcionat per JADX pot variar respecte l'original, i en aquest cas es pot apreciar una diferència relativament gran. En la funció *onViewCreated* es crea un *setOnClickListener*, però en lloc d'especificar la funcionalitat dins d'aquest, s'ha mogut a una nova funció lambda dins de LoginFragment. El resultat és que la funcionalitat del botó està separada de la creació del botó. Tot i això, es veu ràpidament que la funcionalitat és la mateixa que la versió original, creant una instància de JSONController i cridant la funció *checkPassword*. El nom de les funcions no ha sigut alterat.

Observant ara la versió obfuscada, veiem que aquí sí s'aprecia l'efecte que ha tingut el ProGuard. Com la vegada anterior, ara les funcions de la classe LoginFragment han canviat de nom i, tornant a comparar amb la versió original, concloem que la funció *onViewCreated* ara s'anomena *C*. En aquesta funció *C* també es crea un *setOnClickListener*, però la funcionalitat no està especificada en una funció lambda com abans, sinó que està en una classe anomenada *d1.c* apart de LoginFragment. Si analitzem aquesta classe, té una funció *onClick*, i es pot reconèixer que aquesta és la funcionalitat de *checkPassword*. Tot i això, ni el nom de les variables ni els Strings han sigut modificats, tot i que l'estructuració de codi sí s'ha vist modificada respecte la versió original.

Un cop analitzades les diferències entre haver utilitzat ProGuard i sense haver-lo utilitzat, podem veure que les alteracions bàsiques del ProGuard ajudarien a confondre a un atacant poc experimentat, ja que, depenent de com estigui implementat el codi, el resultat pot ser un codi que no deixi molt clar què està fent a cada lloc. Tot i això, es veu clarament que això no impediria un atac si es tingués la pràctica i la paciència suficient, ja que fins i tot en el cas del FirstFragment es fa més obvi encara on es fa la crida a la funció de la classe obfuscada. A més a més, al no haver aplicat

cap modificació ni a noms de variables ni als Strings, es dona molta informació sobre el que s'està fent en cada part, tot i que els noms de les funcions si hagi sigut modificat. Clarament a la línia de codi que es declara un String amb valor *Incorrect user or password*, s'està fent una validació d'usuari.

Per tant, si l'objectiu que volem aconseguir és impedir o, com a mínim, dificultar considerablement un atac d'enginyeria inversa, haurem d'aplicar més mètodes d'obfuscació que modifiquin els Strings.

4 APLICACIÓ D'OBFUSCACIÓ DE STRINGS

Fins ara, l'únic mètode que s'ha aplicat a nivell d'obfuscació es el ProGuard. Aquest es limita a canviar noms de funcions i a reestructurar algunes parts de codi, però no es fa cap mesura ni sobre els noms ni els valors de les variables que s'utilitzen.

La variable més vulnerable que es pot trobar en el codi són els Strings, degut a que poden descobrir la funcionalitat que està fent el codi, estalviant molt temps a un atacant a investigar els fitxers protegits per ProGuard. Per tant, en aquest apartat es mostra com aplicar tres obfuscadors diferents que s'encarreguen d'ocultar el contingut dels Strings amb mètodes diferents. Els tres són a nivell de compilació, dos per a Kotlin i un per a Java.

A la secció B.2 de l'apèndix es mostren els efectes que tenen aquests obfuscadors sobre els Strings de la funció *initFiles* de la classe JSONController.

4.1 Paranoid

Paranoid [6] és un obfuscador de Strings per a aplicacions Android programades en Kotlin. Aquest, de la mateixa manera que els altres obfuscadors, s'implementa a la nostre aplicació a través de la configuració de Gradle [7]. Gradle és l'eina de construcció de projectes de software, la qual s'encarrega de gestionar configuracions de l'aplicació, ja sigui versions per impedir incompatibilitats entre diferents funcionalitats, dependències de funcionalitats que pot haver-hi en diferents parts del projecte, entre moltes d'altres. Per exemple, la declaració de si es vol utilitzar ProGuard en la compilació o no, s'indica en el fitxer de Gradle.

Per començar a tractar amb Gradle, primer s'ha de fer una distinció entre els fitxer que ens afecten principalment, el *build.gradle* del projecte i el *build.gradle* de l'aplicació (mòdul). Gradle permet especificar configuracions a nivell de projecte o a nivell de mòdul. Aquests mòduls es poden considerar com a subprojectes, per tant, si es tinguessin diferents mòduls independents dins del mateix projecte, es gestionarien independentment a partir dels seus fitxers Gradle de mòdul (un per a cadascun), però si es volgués aplicar una configuració a totes alhora, es faria a nivell de projecte. La primera característica que s'ha d'indicar en el *build.gradle* del projecte és declarar quin *classpath* es necessita per utilitzar l'obfuscador, el del plugin de Paranoid. A més a més, s'haurà de concretar des de quin *repository* es descarregarà els *classpaths* especificats, en el cas de Paranoid el de *mavenCentral*.

A nivell de mòdul, s'ha d'especificar que es vol utilitzar el plugin de Paranoid (*io.michaelrocks.paranoid*). Les configuracions per defecte permetran que s'apliqui l'obfuscació a totes aquelles classes que s'anotin amb un *@Obfuscate* a la seva declaració. Per a poder-les obfuscar, s'haurà d'importar la classe *Obfuscate* del plugin declarat.

En el cas de la nostra aplicació, es vol obfuscar la classe JSONController, ja que en aquesta classe es troba la funcionalitat de lectura i escriptura de fitxer i gestió de variables JSON, per tant, hi ha Strings que són importants obfuscar. Si es compila la APK i s'analitza de nou el resultat amb JADX, es pot veure instantàniament canvis respecte la versió anterior.

Els noms dels fitxers, que abans estaven declarats en Strings en clar, ara s'han substituir per una crida a una funció d'una nova classe *Deobfuscator*. Si s'analitza aquesta nova funció, es veu que es dirigeix a una altra classe, i aquesta fa una nova operació molt complicada per a recuperar el String original.

Com es pot veure, tècnicament és possible recuperar el text original, ja que descompilant es poden obtenir totes les eines necessàries per desfer el procés d'obfuscació (és lògic, ja que l'aplicació necessita una manera de desfer també el procés). Tot i això, si la magnitud de l'aplicació fos suficientment gran, i tenint en compte que podem utilitzar el ProGuard alhora, es veu que el temps que hauria de dedicar un atacant que no conegués el codi seria molt més gran que si no s'hagués pres cap mesura de seguretat.

4.2 Obfustring

Obfustring [8] també aporta la funció d'obfuscar els Strings. Per a poder aplicar aquest altre obfuscador, s'han de seguir els mateixos passos que el cas anterior. El nous *classpath* a especificar és *io.github.c0nnor263.obfustring-plugin*, i s'obté d'una URL especifica de *maven* [9]. Posteriorment, en el mòdul s'ha d'especificar que aquest plugin serà una dependència a tenir en compte, i ja es podrà indicar quina classe es vol obfuscar amb el marcador *@ObfustringThis*.

Si s'analitzen els resultats amb JADX, es pot observar que els Strings han sigut substituïts per crides a una funció amb nom *m18vimpl$default* d'una nova classe *ObfStr*. Aquesta classe referència una nova funció anomenada *m17vimpl* i, si s'analitza aquesta, es pot observar que serà molt més complicat desfer l'operació que el cas de Paranoid. La funció està plena de salts de condicions, fent molt complicat seguir el flux d'execució que seguiria la funció. Per tant, si es pretén analitzar la funcionalitat, s'haurà de dedicar una quantitat de temps astronòmica.

4.3 Enigma

Aquest obfuscador, Enigma [10], té un comportament diferent als casos anteriors. Només funciona per a classes Java, per tant, per a comprovar els resultats, s'ha creat una aplicació amb una funció molt reduïda del cas original, però en Java en lloc de Kotlin. Simplement llegeix un fitxer quan s'inicialitza l'aplicació.

A l'hora de la configuració del Gradle, el nou valor del *classpath* és *gradle.plugin.chrisney:enigma:1.0.0.8*, i la localització es una URL concreta amb plugins de *maven* [9]. A nivell de mòdul, s'ha d'afegir el plugin

com.chrisney.enigma. Amb la configuració per defecte d'aquest plugin ja es pot compilar la APK.

Un cop s'inicialitza la compilació, ja es poden observar els canvis respecte els obfuscadors anteriors. Per començar, la compilació comença creant una copia de seguretat de tots els fitxers del directori de treball en una nova carpeta. Posteriorment encripta els fitxers, aplica la compilació sobre els fitxers encriptats i desfà l'encriptació. Per tant, totes les modificacions aplicades afecten el resultat de la APK, però el programador no treballa mai amb els fitxers encriptats.

Si s'analitza els resultats en JADX, seguint tota la seqüència de funcions creada pel ProGuard, s'arriba a la funció de *initFiles*, que llegeix d'un fitxer per a escriure els continguts a un nou fitxer. Es pot veure que els Strings estan en forma d'*arrays* de bytes, per tant, s'ha de dedicar una quantitat de temps considerable per a veure quin valor conté cada String. Tot i això, com en els casos anteriors, si un atacant aconsegueix arribar a entendre com estan organitzats els nous Strings, podria desfer l'obfuscació i obtenir el valor inicial.

5 FORMACIÓ D'ENGINYERIA INVERSA

Si es té com a objectiu aplicar mesures de seguretat per a protegir una aplicació de possibles atacs d'enginyeria inversa, és necessari tenir uns conceptes bàsics de què pot fer una persona maliciosa per tal d'obtenir informació o recursos no desitjats.

Per tal de dur a terme una investigació més pràctica, s'ha completat un curs online gratuït [11] que explica els aspectes principals de l'enginyeria inversa, començant per quines eines i softwares s'utilitzen, com fer-ne us d'elles, i finalment el procés que es segueix per a trobar vulnerabilitats en l'aplicació.

5.1 Concepte de codi Smali

La compilació de codi Java normalment es duu a terme utilitzant JVM (Java Virtual Machine) [12], però si es tracta d'una aplicació d'Android, s'utilitzava Dalvik VM, una màquina virtual optimitzada per a l'ús de memòria, bateria i rendiment, aspectes vitals per a un telèfon mòbil. Posteriorment es va passar a utilitzar ART (Android RunTime). El mètode que seguieix Android per compilar l'aplicació és transformar totes les classes dins del fitxer *.dex* que es troba dins de les APKs, i utilitza el llenguatge assemblador Smali. Això implica que partint d'una APK, si volem alterar el seu comportament, haurem de fer el procés de backsmaling (aconseguir el codi Smali a partir del *.dex*) i ésser capaços d'entendre i modificar el codi Smali, ja que aquest és el que s'utilitza per definir funcionalitats dins l'aplicació.

El codi Smali utilitza registres de 32 bits que permeten utilitzar tots els tipus de variables. L'objectiu d'aquest treball no és explicar la sintaxis [13] d'aquest, però comparteix molts aspectes amb ASM, llenguatge de baix nivell que també utilitza registres, estudiat a la carrera.

5.2 Software d'utilitat

Les eines necessàries per a dur a terme un atac d'enginyeria inversa es podrien reduir a una que et permetés modificar el codi Smali i re-compilar-lo. Tot i això, resulta molt difícil entendre a simple vista aquest codi, degut a que les variables no tenen nom, el codi s'ha de comprendre a partir dels registres usats. Per tant, és molt convenient l'ús d'eines extres que permetin un anàlisi previ del codi en forma de Java/Kotlin, contrastar-lo amb el codi Smali i aplicar les modificacions necessàries en les ubicacions desitjades.

La primera eina que es pot utilitzar es JADX, eina que ja ha sigut mencionada anteriorment en aquest informe. JADX permet, a partir d'una APK, analitzar totes les classes i fitxers que es troben dins l'aplicació. Permet analitzar el codi en forma de llenguatge d'alt nivell, ja sigui Java, Kotlin o l'utilitzat, però també permet analitzar la versió Smali. No obstant, JADX no permet la funcionalitat de re-compilar el codi, per tant, JADX es limitarà a ser exclusivament una eina d'anàlisi de codi.

Un altre software que s'utilitzarà únicament per a analitzar codi serà GDA [14]. Aquest presenta una utilitat molt superior a les demés quan es tracta de fer un anàlisi superficial sobre tota l'aplicació. Aquesta eina mostra el percentatge d'ús de cada tipus de variables, proporciona eines que permeten analitzar el codi per veure si reconeix patrons de *malware*, però sobretot proporciona fitxers extres que mostren tots els Strings i totes les URLs que hagin escrites a l'aplicació. Aquest software demostra la importància que tindrà en un futur obfuscar els Strings de la nostra aplicació per tal de no mostrar informació important del comportament del codi de manera no intencionada.

Per acabar, serà necessària la utilització de APKEasyTool. Aquesta és l'eina que ens permetrà fer tant la descompilació de l'aplicació com la recompilació dels fitxers del codi Smali que haguem modificat. A més a més, aquesta eina té automatitzada la re-signatura de l'aplicació, per tant no ens haurem de preocupar d'aquesta.

Com a nota extra, pot ser que no totes les funcionalitats estiguin implementades en les classes Java o Kotlin, sinó en llenguatges com C++. Si aquest fos el cas, un cop realitzada la compilació, aquestes funcionalitats es veurien dins de llibreries en forma de *.so* (Shared Objects) dins la APK. Si necessitéssim modificar aquest codi, podríem utilitzar una eina com Ghidra [15], un descompilador de codi natiu.

5.3 Continguts d'una APK

Si canviem l'extensió d'un fitxer *.apk* a *.rar*, degut a que ambdós es comporten de la mateixa manera, formant part d'un conjunt diferent de fitxers, el podem exportar de la mateixa manera. A continuació es fa una explicació dels fitxers més importants que formen una APK.

A la carpeta de META-INF podem trobar fitxers com els certificats de firma de l'aplicació per a verificar la seva autenticitat. Les carpetes d'*assets* i *res* contenen diferents arxius amb funcionalitats generals del *frontend*. La carpeta de Kotlin conté els fitxers necessaris per a poder fer funcionar les aplicacions programades en Kotlin i no Java. Finalment, el fitxer *classes.dex* conté totes les classes dins del programa.

Si utilitzem l'analitzador de fitxers d'Android Studio podem observar l'interior del fitxer *.dex*. En el cas de la nostra aplicació concreta podem trobar-hi classes relacionades amb funcionalitats Android, Google, Java i Kotlin genèriques que permeten el funcionament correcte de l'aplicació. També hi trobem les classes que s'han programat

agrupades en el directori *com*, juntament amb components de Google que permeten la utilització d'elements com botons, transicions i quadres de text. El directori *com* és el que tindrà més importància a l'hora de fer atacs d'enginyeria inversa.

5.4 Seqüència d'atac senzill

Els passos a seguir per a realitzar un atac d'enginyeria inversa sobre una aplicació son els següents:

- Investigar aplicació
- Anàlisi superficial amb GDA
- Localització dels punts vulnerables amb JADX
- Modificació del comportament amb APKEasyTool

Per començar a trobar possibles punts d'explotació d'una aplicació, no és necessari l'ús de cap eina mencionada en el apartat anterior. Si es fa ús de l'aplicació durant una estona, es podran descobrir components comuns, com per exemple camps de text o botons. Es pot analitzar el comportament d'aquests botons, fixar-se en si mostren missatges de text per pantalla o provoquen una transició. Es pot veure si els camps de text tenen paraules o frases d'ajuda, per a saber el contingut que s'hauria d'introduir. Si s'utilitzés l'aplicació com un usuari normal, aquests aspectes no tindrien massa rellevància. No obstant, com que s'està actuant com a agents maliciosos, es pot utilitzar aquesta informació per a posteriorment vincular comportaments que es vegi en el codi amb conseqüències que causen en l'aplicació final.

Un cop s'ha acabat el procés previ de familiarització amb l'aplicació, es comença fent un anàlisis superficial de tota l'aplicació amb l'eina de GDA. Amb aquesta, es mira a la llista generada de tots els Strings a veure si es pot observar algun text que pugui donar una pista sobre on es troba un punt feble que s'hagi trobat. Si es pren com a exemple l'aplicació utilitzada en aquest treball, se sap que en introduir un usuari i contrasenya incorrectes es mostra per pantalla un missatge d'error. Si es busqués aquest missatge a GDA i es trobés (això implicaria que els Strings no estarien obfuscats), es podria trobar directament a quina part del codi es gestiona el codi d'error i, probablement, la part del codi que es dediqui a comprovar el login.

Un cop es tingui un possible punt d'explotació en el codi, es passaria a JADX, ja que permet comparar ràpidament codi Java/Kotlin amb la versió Smali. Si es trobés el punt on fer una possible modificació, es procediria a mirar on s'hauria d'aplicar dins del codi Smali, es descompilaria l'aplicació amb APKEasyTool, es farien els canvis en el codi i es re-compilaria la APK.

6 Execució d'atacs d'Enginyeria Inversa

En aquest punt es té un conjunt de APKs, algunes protegides amb unes mesures i altres no protegides. Juntament amb els coneixements mostrats a la secció anterior, es pot fer una seqüència d'atac a les APKs per comprovar quines dificultats pot trobar un atacant maliciós i quins punts febles s'haurien de protegir per a millorar la seguretat de l'aplicació.

L'objectiu dels atacs serà aconseguir que es verifiqui l'intent de login sense introduir un usuari i contrasenya incorrectes.

A l'apartat B.3 de l'apèndix es mostren seccions de les funcions que utilitzen els obfuscadors per a desfer l'operació aplicada per obtenir el String original. Analitzant aquestes funcions podrem extreure conclusions sobre si un atacant podria recuperar els Strings originals fàcilment o no.

6.1 Atac sobre una APK desprotegida

Seguint els passos explicats en una seqüencia d'atac 5.4, primer s'ha de fer una inspecció superficial de la aplicació. Si s'interactua amb ella, veiem que els camps tant de registre com de login tenen indicat el nom del valor que s'ha d'introduir. Per tant, es pot deduir que si s'ha de reconèixer algun punt del *frontend*, es pot intentar fer-ho a partir d'aquests components de text. A més a més, en el cas de prémer els botons, apareixen notificacions amb un text que explica quin cas ha ocorregut en l'execució.

Un cop s'ha finalitzat l'anàlisi inicial, cal començar a fer una inspecció sobre el codi de l'aplicació. El software adient per a fer-la és el GDA. Utilitzant aquesta eina es pot consultar els documents que proporciona, amb el conjunt de tots els Strings que hi ha ubicats a l'aplicació. Com que s'ha fet un anàlisi de tota l'aplicació, volem començar comprovant si s'ha trobat un String que coincideixi amb les notificacions que hem trobat a l'hora d'utilitzar l'aplicació. Si es fa una cerca sobre el document *AllStrings* proporcionat pel software del text 'Incorrect user', en efecte apareix una entrada amb el text de la notificació que apareix quan la verificació de login falla. Fent una referència creuada sobre aquesta entrada, es veu que aquest String és creat en una funció anomenada *OnViewCreated* de la classe LoginFragment. Un cop s'ha ubicat la declaració d'un String d'interès, es comença a analitzar el funcionament del codi.

Per a una comoditat superior a l'hora d'analitzar el codi Kotlin, s'utilitzarà JADX, degut a que facilita la comparació simultània entre codi Kotlin i codi Smali. A més, JADX també proporciona eines d'exploració superiors a GDA, un sistema de cerca més complet, que permet la cerca de caràcters en classes, comentaris, o en parts de codi específiques, cosa que pot servir d'ajut a l'hora de buscar funcionalitats concretes. Gràcies a la utilització de GDA se sap la part de codi d'interès, la classe *OnViewCreated* de la classe LoginFragment. GDA també permet veure a quin directori està aquesta classe, per tant trobar la ubicació no es cap problema.

Si es fa una lectura d'aquesta funció, es veu que s'assignen dues variables anomenades 'usuari' i 'contrasenya', per tant, es dedueix que aquests són els valors dels quadres de text de la vista del fragment de LoginFragment. Posteriorment es fa una crida a una funció anomenada *checkPassword* que transmet per paràmetre aquests dos valors, per tant, es conclou que en aquesta funció es fa la comprovació de si els usuaris i contrasenyes son correctes. Aquesta funció retorna un booleà, el qual indica quin cas de la funció s'haurà d'executar posteriorment a la verificació. Es veu que si el booleà retorna *true*, executa la part del codi quan s'han introduït les dades correctament. Per tant, si es vol que s'executi aquesta part de codi quan s'introdueix una resposta incorrecta, cal canviar la comparació del codi

Smali.

Transicionant a la versió Smali que proporciona JADX, es pot buscar la funció *checkPassword* que s'ha vist que era la clau per decidir quin cas ha ocorregut. Si s'analitza el codi, es veu que el resultat de la crida d'aquesta funció s'emmagatzema en un registre anomenat *v6*, i posteriorment es fa una comparació de si el valor és igual a zero. Fent això el codi està comprovant si el valor de *v6* és *true*, que implicaria que a *v6* hi ha el valor 1, o *false*, que hi hauria el valor 0. Per tant, si el valor es *false*, fa un salt de condició. Per assegurar-nos de que no hi ha cap salt condicional, ja que volem executar el cas on s'ha validat la contrasenya, s'ha d'invertir aquesta comparació, fent que es comprovi si el valor del registre *no* és igual a zero. Canviem el *if-eqz* a *if-nez*.

Ara que està localitzada la línia de codi concreta a canviar, s'ha de descompilar la APK utilitzant APKEasyTool, degut a que JADX no permet la modificació dels fitxers. Un cop feta la descompilació, el resultat es troba en el directori dedicat a les APKs descompilades. Si s'accedeix al directori, es veurà que hi ha una carpeta anomenada *smali*, i dins d'aquesta hi ha l'estructura de fitxers que hem anat veient tant a GDA com a JADX. Com que ja se sap la localització de la línia de codi a canviar, es pot alterar el text de Smali amb qualsevulla eina editora de texts. Si es vol assegurar que el nou comportament és el desitjat, es pot alterar també el text de *Login successful*. Quan s'han aplicat totes les modificacions en el codi Smali, seguint utilitzant APKEasyTool, es re-compila la APK indicant aquest nou directori alterat. APKEasyTool generarà la APK desitjada en el directori dedicat, fent les operacions necessàries sobre la signatura de la APK. Aquesta APK pot ser instal·lada en un dispositiu Android, però per analitzar els resultats s'ha utilitzat l'emulador proporcionat per Android Studio.

Un cop instal·lada la nova APK, es pot comprovar que en efecte les modificacions han sigut aplicades correctament. Introduint qualssevol valors, al prémer el botó de login surt la notificació de que la verificació s'ha produït correctament. Cal notar que si en algun moment s'introduïssin un valor d'usuari i contrasenya i el missatge fos que no s'ha pogut fer el login, significaria que s'ha trobat un usuari real. Això és degut a que no s'ha fet que sempre succeeixi la verificació correcta, sinó que s'han intercanviat les comparacions.

6.2 Atac sobre APK amb ProGuard

El procés d'atac sobre una aplicació protegida amb ProGuard segueix majoritàriament el mateix que si no s'hagués protegit la APK. Això és degut a que no hi ha cap protecció sobre els Strings i la diferència més notable és el canvi de nom a les funcions. Això no hauria de crear una dificultat massa gran a un atacant experimentat, ja que tinguessin el nom que tinguessin les funcions, l'atacant no les coneixeria. El factor que més pot dificultar aquest procés és alguna possible optimització que hagués fet el *minifyer* que impliqués ajuntar funcions o reordenar-les.

Es pot trobar de la mateixa manera el punt en el codi on es troba l'operació de validació de l'usuari. En aquest cas, la funcionalitat està implementada a una classe anomenada *d1.c*, però es troba amb la mateixa facilitat a partir de GDA. Un cop es passa a JADX, es troba fàcilment que es fa una comparació abans de mostrar per pantalla el missatge. Per tant s'ha de modificar un altre cop la comparació de *if-eqz* a *if-nez* i re-compilar amb APKEasyTool. Al reinstal·lar la APK a l'emulador, les modificacions s'han produït correctament.

Tot i que la modificació s'ha produït correctament, ja que només s'ha hagut de reconèixer la comparació desitjada, si s'analitza com s'aconsegueix el valor emmagatzemat a la comparació és molt més complicat de comprendre que la versió sense protegir. En el primer atac, es cridava directament a la funció *checkPassword* i amb aquest booleà es feia la comparació. En aquesta APK, s'assigna el valor de l'usuari i contrasenya a un registre i, de manera molt rebuscada, canvien de tipus de variable i s'intercanvien de registre, fins a un punt on es comparen amb el valor desitjat. Si en qualsevol moment la verificació és incorrecta, fa una serie de salts condicionals que acaben arribant a la comparació. Com que el booleà que s'ha utilitzat comença tenint assignat un 0, la verificació falla. Si en tot moment s'ha anat complint la verificació, arriba un punt on s'assigna al booleà un 1. Per tant, tot i que no ha sigut necessària la comprensió de tot aquest procés, demostra que si el ProGuard es complementa amb altres mesures de seguretat s'obtindrà un codi molt més robust.

6.3 Atac sobre APK obfuscada

Com s'ha vist fins ara, els processos han sigut molt senzills perquè s'ha partit de tot moment de saber que existeix un String amb valor *Login succesful!*. Tot i això, a l'hora de fer els atacs sobre les APKs obfuscades no es té aquest coneixement. Si es fa una cerca a GDA no surt cap String d'interès amb la paraula *Login*. Introduint altres paraules que podrien donar una empenta a l'hora de començar a analitzar el codi, tampoc no es troba res. Si busquem els Strings *Username* i *Password* no trobem cap referència útil, tot i que sí existeixen variables amb aquest nom. No apareixen a la cerca gràcies a l'efecte del ProGuard.

Per continuar a buscar possibles punts d'entrada, es passa directament a JADX. El següent pas que es pot seguir és anar al directori per defecte que genera Android Studio per a aplicacions, el directori *Com*. En examinar aquest directori, només es troben 2 directoris més, per tant, ràpidament es troba el conjunt de classes que s'han programat per a la funcionalitat de l'aplicació. Seguint analitzant, i veient que una té la funcionalitat del *Login*, es pot examinar i es veu que té els Strings obfuscats, provocant que no es pugui saber quin valor tenen. En el constructor d'aquesta classe es crea un botó, per tant es pot suposar que aquest botó és el que crida l'execució del procés de validació. Analitzant la funcionalitat del botó, es troba que després de condicions es creen components de *Toast* amb un String obfuscat passat com a paràmetre. El component de *Toast* són les notificacions temporals que apareixen amb notificacions a les aplicacions. Com que no es coneixen els valors dels Strings, no se sap quin es cada cas, però es pot intuir que si s'inverteix la condició s'aconseguirà el resultat. No obstant, no només hi ha 2 *Toasts*, n'hi ha 5. Això és degut a que s'han fusionat les funcionalitats de Login i Registre. Fent prova i error, es poden invertir totes les comparacions prèvies a la creació dels *Toasts* fins a trobar la comparació desitjada.

Com es pot veure, l'obfuscació no ha impedit que es pu-

gui complir l'objectiu de l'atac. Cap dels tres obfuscadors ho ha pogut impedir. Tot i això, no vol dir que els obfuscadors no hagin sigut d'utilitat. L'única raó per la que s'ha pogut continuar l'atac després de no haver trobat cap resultat amb GDA ha sigut perquè el codi estava en el directori per defecte. Si el nom del directori fos diferent, no s'hagués progressat amb tanta facilitat. A més a més, els *Toasts* han tingut un paper molt important a l'hora de diferenciar quins punts eren els més probables a ser vulnerables. Per tant, es pot considerar crear una funció que crei les notificacions. Si aquest fos el cas, entre els obfuscadors de Strings i el ProGuard ja es crearia una barrera de continuació molt més alta.

7 RESULTATS

Davant qualsevol dels tres obfuscadors, el mateix atac ha tingut èxit. Això és degut a que els obfuscadors no alteren el flux d'execució, només l'aparença dels Strings. Per tant, el factor més diferenciador és la manera en la que un atacant podria intentar desfer l'obfuscació per obtenir el valor original del String.

Si s'aconsegueix arribar fins la funció de des-obfuscació de Paranoid, es troba una funció complicada a resoldre per una persona, però senzilla d'entendre. No té cap salt condicional i només es fan operacions matemàtiques, fent que una persona experimentada pugui crear una funció que desfaci l'obfuscació de manera ràpida. Si de la mateixa manera s'arriba a la part de codi que desfà l'obfuscació de Obfustring, es troba un procés de recuperació molt més enrevessat. Hi ha molts salts condicionals i operacions confuses. El temps que s'hauria de dedicar per aconseguir recuperar el String inicial és molt elevat. Finalment, l'obfuscador Enigma, que només serveix per a codi Java, també oculta els valors dels Strings amb un procés molt més complicat. Com que utilitza encriptació i genera fitxers auxiliars, amb l'ajut de ProGuard es crea una funció pràcticament impossible de desfer. No obstant, cal notar que moltes vegades, al compilar la APK han sorgit problemes quan s'utilitzava Enigma. Degut a que encripta els Strings, compila, i després els desencripta, el procés de compilació segueix fent operacions ocultes al programador. Si mentre segueix fent el procés de recuperació de la versió inicial el programador fa algun canvi i re-compila de manera ràpida, l'execució fallarà perquè el procés seguia ocupat i avortarà, provocant que el programador quedi amb el codi encriptat. S'hauria de fer el procés de recuperació manualment.

Degut a tots els aspectes mencionats anteriorment, es decideix que la millor opció és la del Obfustring.

8 MÈTODE D'IMPLEMENTACIÓ DE OBFUSTRING

Si es volgués aplicar l'obfuscador Obfustring a un projecte d'una aplicació d'Android, s'han de seguit el següent procediment:

Al fitxer *build.gradle* del projecte (figura 2):

- Especificar la localització del *repository*
- Especificar el *classpath*

```
buildscript {
    repositories {
        google()
        //noinspection JcenterRepositoryObsolete
        jcenter()
        // Add Maven repo
        mavenCentral()
        maven {
            url "https://plugins.gradle.org/m2/"
        }
    }
    dependencies {
        //noinspection AndroidGradlePluginVersion
        classpath 'com.android.tools.build:gradle:3.5.2'
        classpath "io.github.c0nnor263:plugin:11.09"
    }
}

plugins {
    id 'com.android.application' version '7.3.1' apply false
    id 'com.android.library' version '7.3.1' apply false
    id 'org.jetbrains.kotlin.android' version '1.7.10' apply false
}
```

Fig. 2: Modificacions build.gradle del projecte

Al fitxer *build.gradle* del mòdul amb les classes que es vulguin obfuscar:

- Introduir el plugin corresponent
- Declarar la variable *packageKey* (figura 3)
- Especificar la dependència (figura 4)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'io.github.c0nnor263.obfustring-plugin'
}

obfustring{
    packageKey = "com.conboi.myapplication" // comDconboinmyapplication
}
```

Fig. 3: Modificacions build.gradle del mòdul

```
dependencies {
    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.4.1'
    implementation 'com.google.android.material:material:1.5.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    implementation 'androidx.navigation:navigation-fragment-ktx:2.4.1'
    implementation 'androidx.navigation:navigation-ui-ktx:2.4.1'
    implementation "io.github.c0nnor263:obfustring-core:11.09"
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

Fig. 4: Declaració de dependència en el mòdul

A les classes del mòdul que es volen obfuscar:

- Anotar les classes amb *@ObfustringThis* (figura 5)
- Importar la classe necessària per aplicar correctament l'anotació anterior

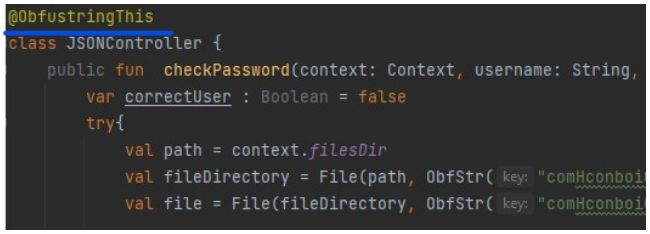

Fig. 5: Anotació de les classes

9 Conclusions

En conclusió, l'ús d'un obfuscador no és mesura de seguretat suficient per a impedir un atac d'enginyeria inversa, però segueix essent una ajuda imprescindible. La facilitat amb la que es poden fer atacs d'enginyeria inversa sobre una aplicació amb Strings en clar és molt notable. Per tant, aplicar obfuscació sobre els Strings augmentarà considerablement el temps que haurà de dedicar un atacant per obtenir alguna informació important.

Tot i això, a mesura que s'han anat executant els tests dels diferents obfuscadors, s'han trobat moltes mesures senzilles que es poden aplicar a les aplicacions que també afegirien una capa de protecció sense dedicar-hi molt esforç, afegint encara més temps que hauria de dedicar un agent maliciós a trobar informació sobre el codi. La utilització de ProGuard, que alterava els noms de les variables i, en alguna ocasió, reordenava l'ordre d'execució d'algunes funcions, es combina perfectament amb la dels obfuscadors de Strings. Per tant, es recomana que es segueixi aplicant, tot i que inicialment semblés que el seu efecte no era excessivament positiu. Una altre mesura a tenir en compte que es podria aplicar, seria tenir un sistema de fitxers que no segueixi la versió generada per defecte pel nostre programa utilitzat per generar l'aplicació, en aquest cas Android Studio. Aquesta mesura es recomanada no només perquè ajudaria als propis programadors a segmentar les funcionalitats del codi i tenir una aplicació més organitzada, però també perquè molts dels atacs ens hem basat en que part del codi útil estaria en directoris específics, ja que son els directoris que es creen per defecte. Si al intentar fer un atac d'enginyeria inversa no es troba cap directori generat per defecte, començar a fer l'atac serà molt més complicat. Per últim, s'ha vist que un aspecte que també pot perjudicar a l'hora de protegir l'aplicació és la generació de notificacions dins l'aplicació. Això es podria solucionar parcialment creant una funció específica que generés les notificacions, i juntament amb l'obfuscació de Strings com l'ús del ProGuard, dificultar la comprensió d'on es generen aquestes notificacions en el codi.

Referències

[1] Alex Marin, *GitHub*, 2022 [En línia]. Disponible:
https://github.com/Alex1527272/LogIn

[2] Android per a desenvolupadors, *Build Your First Android App in Kotlin*, 2022 [En línia]. Disponible:
https://developer.android.com/codelabs/build-your-first-android-app-kotlin

[3] Android per a desenvolupadors, *Configure your build*, [En línia]. Disponible:
https://developer.android.com/studio/build/index.html

[4] Android per a desenvolupadors, *Shrink, obfuscate, and optimize your app*, [En línia]. Disponible:
https://developer.android.com/studio/build/shrink-code

[5] Skylot, *GitHub JADX*, [En línia]. Disponible:
https://github.com/skylot/jadx

[6] M. Rocks, *GitHub Paranoid*, [En línia]. Disponible:
https://github.com/MichaelRocks/paranoid

[7] Gradle, *Gradle User Manual*, [En línia]. Disponible:
https://docs.gradle.org/current/userguide/userguide.html

[8] c0nnor263, *GitHub Obfustring*, [En línia]. Disponible:
https://github.com/c0nnor263/obfustring-plugin

[9] Gradle, *Plugins gradle*, [En línia]. Disponible:
https://plugins.gradle.org/m2/

[10] C. Ney, *GitHub Enigma*, [En línia]. Disponible:
https://github.com/christopherney/Enigma

[11] FreeCourseSites, *Android Reverse Engineering*, [En línia]. Disponible:
https://freecoursesites.org/android-reverse-engineering

[12] GeeksForGeeks, *Compilation and Execution of a Java Program*, [En línia]. Disponible:
https://www.geeksforgeeks.org/compilation-execution-java-program/

[13] Programmer All, *Smali Grammar Basics*, [En línia]. Disponible:
https://www.programmerall.com/article/9200827288/

[14] Charles2Gan, *GitHub GDA*, [En línia]. Disponible:
https://github.com/charles2gan/GDA-android-reversing-Tool/releases

[15] N. S. Agency, *GitHub Ghidra*, [En línia]. Disponible:
https://github.com/NationalSecurityAgency/ghidra/releases

APÈNDIX

A.1 Especificacions de l'aplicació

A.1.1 Diagrama de classes

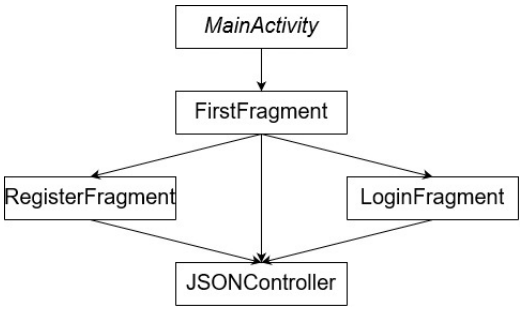

Fig. 6: Classes de l'aplicació

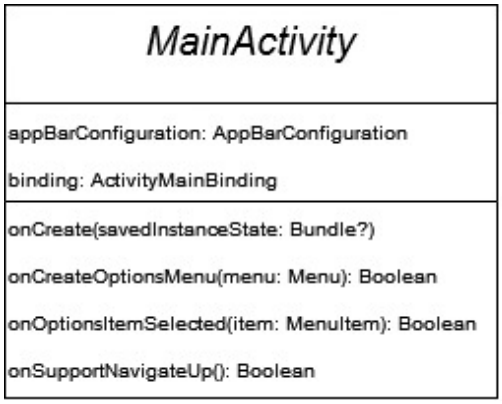

Fig. 7: Classe mainActivity

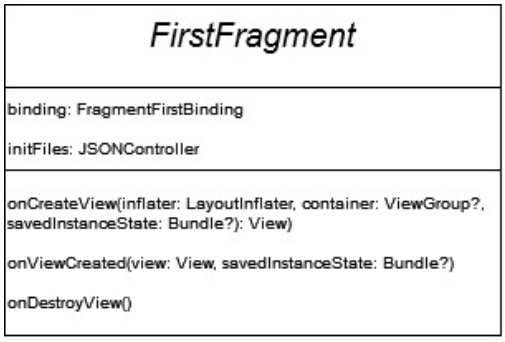

Fig. 8: Classe firstFragment

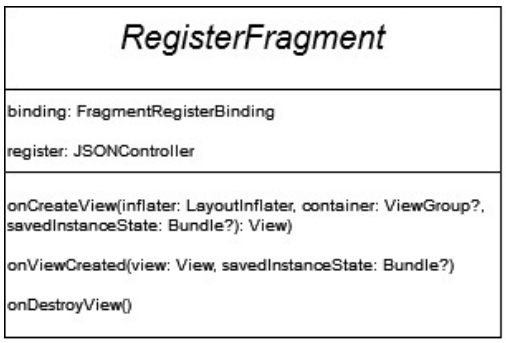

Fig. 9: Classe registerFragment

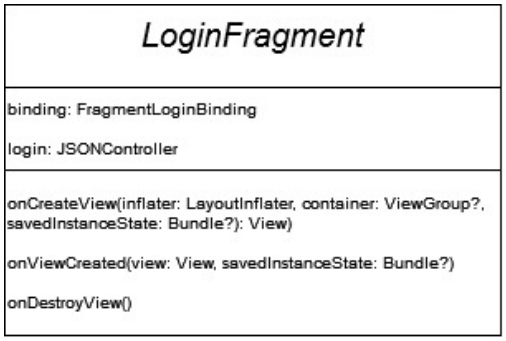

Fig. 10: Classe loginFragment

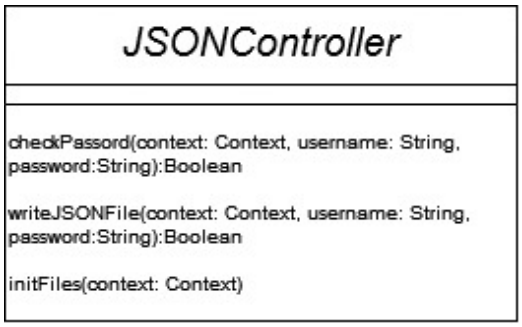

Fig. 11: Classe JSONController

A.1.2 Funcionament de l'aplicació

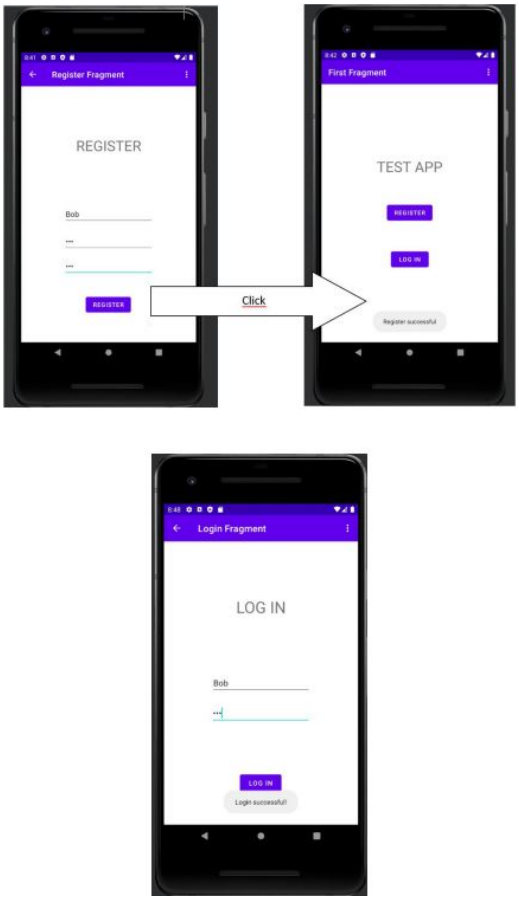

Fig. 12: Funcionalitat de l'aplicació

B.2 Efecte dels obfuscadors

B.2.1 Secció funció *initFiles* sense obfuscador

```
public final void
    initFiles(android.content.Context
    context) {
      kotlin.jvm.internal.Intrinsics
          .checkNotNullParameter(context,
              "context");
      android.content.res.AssetManager
          assets = context.getAssets(); //
          Catch: java.lang.Exception -> L46
      if (assets == null) goto L45;
      java.io.InputStream inputStream =
          assets.open("accounts.json"); //
          Catch: java.lang.Exception -> L46

      ...

}
```

B.2.2 Secció funcionalitat *initFiles* amb ProGuard

```
public final android.view.View
    t(android.view.LayoutInflater
    layoutInflater, android.view.ViewGroup
    viewGroup, android.os.Bundle bundle) {

      ...

      android.content.res.AssetManager
          assets = h3.getAssets(); //
          Catch: java.lang.Exception -> L49
      if (assets == null) goto L48;
      java.io.InputStream inputStream =
          assets.open("accounts.json"); //
          Catch: java.lang.Exception -> L49
  L13:
      if (inputStream == null) goto L47;
      java.io.InputStreamReader
          inputStreamReader = new
          java.io.InputStreamReader(inputStream,
          v2.a.f3727a); // Catch:
          java.lang.Exception -> L49
      if ((inputStreamReader instanceof
          java.io.BufferedReader) == false)
          goto L46;
      java.io.BufferedReader
          bufferedReader =
          (java.io.BufferedReader)
          inputStreamReader; // Catch:
          java.lang.Exception -> L49


      ...

}
```

B.2.3 Secció funció *initFiles* amb Paranoid

```
public final void
    initFiles(android.content.Context
    context) {
      public final void
          initFiles(android.content.Context
          context) {
      kotlin.jvm.internal.Intrinsics
          .checkNotNullParameter(context,
              io.michaelrocks.paranoid
          .Deobfuscator$app$Debug
          .getString(1003457377706031990L));
      android.content.res.AssetManager
          assets = context.getAssets(); //
          Catch: java.lang.Exception -> L46
      if (assets == null) goto L45;
      java.io.InputStream inputStream =
          assets.open(io.michaelrocks
          .paranoid.Deobfuscator$app$Debug
          .getString(1003457343346293622L));
              // Catch: java.lang.Exception
              -> L46

      ...

}
```

B.2.4 Secció funció *initFiles* amb Obfustring

```
public final void
    initFiles(android.content.Context
    context) {
      kotlin.jvm.internal.Intrinsics
          .checkNotNullParameter(context,
              "context");
```

```
        android.content.res.AssetManager
            assets = context.getAssets(); //
            Catch: java.lang.Exception -> L47
        if (assets == null) goto L46;
        java.io.InputStream inputStream =
            assets.open(io.github
           .a26197993b77e31a4
           .o7b471d74a53 ... .ObfStr
           .m18vimpl$default(io.github
           .a26197993b77e31a4
           .o7b471d74a534 ... .ObfStr
           .m16constructorimpl
           ("comKconboiQmyapplication"),
               "cqoswbgt.xayz", false, 2,
               null)); // Catch:
               java.lang.Exception -> L47

        ...

}
```

B.2.5 Secció funció *initFiles* amb Enigma

```
public void
    initFiles(android.content.Context
    context) {
       java.io.InputStream inputStream =
           context.getAssets().open(com
           .chrisney.enigma.EnigmaUtils
           .enigmatization(new byte[]{51,
               -82, -73, 39, -101, -8, 125,
               5, 78, 24, 17, 122, 78, 36,
               -19, -90})); // Catch:
               java.io.IOException -> L15
       byte[] array = new
           byte[inputStream.available()]; //
           Catch: java.io.IOException -> L15
       inputStream.read(array); // Catch:
           java.io.IOException -> L15


       ...

}
```

B.3 Anàlisi dels obfuscadors

B.3.1 Operació d'obfuscació de Paranoid

```
public class DeobfuscatorHelper {
   public static java.lang.String
       getString(long id,
       java.lang.String[] chunks) {
       long state = io.michaelrocks.paranoid
           .RandomHelper.next(io.michaelrocks
           .paranoid.RandomHelper.seed(id &
               4294967295L));
       long low = (state >>> 32) & 65535;
       long state2 =
           io.michaelrocks.paranoid
           .RandomHelper.next(state);
       long high = (state2 >>> 16) &
           (-65536);
       int index = (int) (((id >>> 32) ^
           low) ^ high);
       long state3 = getCharAt(index,
           chunks, state2);
       int length = (int) ((state3 >>> 32)
           & 65535);
       char[] chars = new char[length];
       int i = 0;
   L2:
       if (i >= length) goto L6;
       state3 = getCharAt((index + i) + 1,
           chunks, state3);
       chars[i] = (char) ((state3 >>> 32) &
           65535);
       i = i + 1;
       goto L2
   L6:
       return new java.lang.String(chars);
   }

   ...

}
```

B.3.2 Operació d'obfuscació de Obfustring

```
public static final java.lang.String
    m17vimpl(java.lang.String arg0,
    java.lang.String string, boolean
    encrypt) {

       ...

       L13:
       java.lang.String
           $this$v_impl_u24lambda_u2d52 =
           $this$v_impl_u24lambda_u2d5;
       java.lang.CharSequence
           $this$forEachIndexed$iv2 =
           $this$forEachIndexed$iv;
   L15:
       i = i + 1;
       str = arg0;
       str2 = string;
       index$iv = index$iv2;
       z = z2;
       $this$v_impl_u24lambda_u2d5 =
           $this$v_impl_u24lambda_u2d52;
       $this$forEachIndexed$iv =
           $this$forEachIndexed$iv2;
       goto L5
   L16:
       z2 = z;
       isEscape = false;
   L17:
       z2 = z;
       goto L13
   L18:
       z2 = z;
       int skipSymbolsCount2 =
           skipSymbolsCount;
       if ('A' <= item$iv) goto L21;
   L138:
       boolean z3 = false;
   L23:
       int symbolCodeToAdd = 97;
```

```
      if (z3 == false) goto L40;
      symbolCodeToAdd = 65;
   L26:
      boolean isEscape2 = isEscape;
      if (symbolCodeToAdd != 65) goto L38;
      int i2 = 38;
   L29:
      int decryptInt = i2 + 26;
      if (encrypt == false) goto L37;
      int charAt = ((item$iv +
          str.charAt(keyLoopIndex)) - 90) %
          26;



      ...

}
```

B.3.3 Operació d'obfuscació de Enigma

```
public static java.lang.String
    enigmatization(byte[] enc) {
      byte[] keyValue = keyToBytes(data);
      byte[] result = decrypt(keyValue,
          enc);

      ...

   }

   public static byte[] decrypt(byte[]
       keyValue, byte[] encrypted) throws
       java.lang.Exception {
      javax.crypto.SecretKey skeySpec =
          new javax.crypto.spec
         .SecretKeySpec(keyValue, "AES");
      javax.crypto.Cipher cipher =
          javax.crypto.Cipher
         .getInstance("AES/CBC/PKCS5Padding");

      ...

      return cipher.doFinal(encrypted);
   }

   public static byte[] keyToBytes(int[]
       key) {
      int size = (key.length / 16) * 16;
      java.lang.StringBuilder builder =
          new java.lang.StringBuilder();
      int i = 0;
   L2:
      if (i >= size) goto L6;
      builder.append((char) key[i]);
      i = i + 1;
      goto L2
   L6:
      return builder.toString().getBytes();
   }
```