VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

APLIKACE PRO ANDROID NA BEZPEČNOSTNÍ MONITOROVÁNÍ KOMUNIKACE

ANDROID APP FOR SECURITY MONITORING OF COMMUNICATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. KAROLÍNA KLEPÁČKOVÁ
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2019

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2018/2019

Zadání diplomové práce

22204

Studentka: **Klepáčková Karolína, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Aplikace pro Android na bezpečnostní monitorování komunikace**
Android App for Security Monitoring of Communication
Kategorie: Bezpečnost
Zadání:

1. Seznamte se s problematikou vývoje aplikací pro Android; zaměřte se na síťovou komunikaci.
2. Seznamte se s problematikou bezpečnosti komunikace aplikací pro Android.
3. Vyhledejte a analyzujte dostupné nástroje pro bezpečnostní monitorování komunikace aplikací pro Android.
4. Navrhněte a prototypujte techniky a nástroje pro bezpečnostní monitorování komunikace aplikací.
5. Integrujte techniky bezpečnostního monitorování komunikace aplikací do aplikace s vhodným a použitelným uživatelským rozhraním.
6. Testujte vyvíjenou aplikaci a iterativně ji vylepšujte.
7. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Bill Phillips, Chris Stewart, Kristin Marsicano: Android Programming: The Big Nerd Ranch Guide, 2017, ISBN: 9780134706054
- Jim Doherty: Wireless and Mobile Device Security, Jones & Bartlett learning, ISBN: 978-1-284-05927-4
- Jon Erickson: Hacking, No Starch Press, 2003, ISBN 1-59327-007-0
- Sverre H. Huseby: Zranitelný kód, Computer Press, 2006, ISBN 80-251-1180-6
- Android Developers: https://developer.android.com/index.html

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Herout Adam, prof. Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 7. listopadu 2018

Abstrakt

Tato diplomová práce je zaměřena na implementaci aplikace pro monitorování síťové komunikace ostatních aplikací v mobilním zařízení s platformou Android. Poskytuje uživateli informace o bezpečnostních rizicích, jenž mohou poškodit jeho soukromí či zařízení. Využívá lokální VPN skrze kterou tuneluje veškerá data odesílaná do bezdrátové sítě. Ty lze propojit s aplikací jenž je odeslala díky tomu, že jádro Androidu je odvozeno od Linuxového jádra a lze z něj získat informace o ustanovených síťových připojeních a identifikátoru aplikace přiřazené k danému připojení. Toto mapování umožňuje získat bližší informace o aplikaci, která je pro mobilní zařízení potenciálně nebezpečná. Na základě zjištěných údajů se může uživatel rozhodnout aplikaci dále nepoužívat, či upozornit její tvůrce na případný problém.

Abstract

This diploma thesis is focused on implementation of application for security monitoring of network communication of other applications in mobile device with Android platform. Provides users information about security risks that may harm his/her privacy or device. It uses a local VPN to tunnel all data sent to the wireless network. These can be linked to an application that has sent them because the Android kernel is derived from the Linux kernel and can be used to retrieve information about established network connections and the application identifier associated with the connection. This mapping allows to get more information about an app that is potentially dangerous for your mobile device.

Klíčová slova

bezpečnost Androidu, WEP, WPA, WPA2, WPA3, HTTP, HTTPS, SSL, TLS, certificate pinning, Android, Kotlin, Android kernel, VPN

Keywords

Android security, WEP, WPA, WPA2, WPA3, HTTP, HTTPS, SSL, TLS, certificate pinning, Android, Kotlin, Android kernel, VPN

Citace

KLEPÁČKOVÁ, Karolína. *Aplikace pro Android na bezpečnostní monitorování komunikace.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Adam Herout, Ph.D.

Aplikace pro Android na bezpečnostní monitorování komunikace

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

. . . . . .
Karolína Klepáčková
21. května 2019

Poděkování

Ráda bych poděkovala vedoucímu za vstřícný osobní přístup a cenné rady, které mi v průběhu řešení mé diplomové práce poskytl.

Obsah

1 Úvod **2**

2 Mobilní zařízení a jejich bezpečnostní rizika **3**
2.1 Bezdrátové sítě a jejich bezpečnostní protokoly . . . 3
2.2 Zabezpečení komunikace na úrovni aplikací . . . 9
2.3 RESTful API . . . 14
2.4 Senzory a přístupové technologie . . . 15
2.5 Bezpečnost platformy Android . . . 15

3 Použité technologie **21**
3.1 Operační systém Android . . . 21
3.2 Kotlin . . . 27
3.3 Formát souboru pcap . . . 29

4 Návrh aplikace **31**
4.1 Srovnání již existujících aplikací . . . 32
4.2 Získání procesů a síťových připojení . . . 33
4.3 Zachytávání síťového provozu . . . 34
4.4 Ukládání paketů pro další zpracování . . . 35
4.5 Analýza paketů . . . 36
4.6 Propojení paketů a aplikace . . . 37
4.7 Návrh uživatelského rozhraní . . . 38
4.8 Vylepšení aplikace na základě uživatelského testování . . . 40

5 Implementace **42**
5.1 Architekturní vzory . . . 42
5.2 Navigace a architektura Single Activity . . . 43
5.3 Bypass paketů . . . 45
5.4 Analýza zachycených paketů . . . 46
5.5 Mazání souborů v aplikaci . . . 47
5.6 Výpočetně náročné operace . . . 48

6 Vyhodnocení dosažených cílů **50**
6.1 Možná vylepšení . . . 51
6.2 Nedostatky aplikace . . . 51
6.3 Výsledné statistiky . . . 52

7 Závěr **54**

Literatura **55**

Přílohy **57**

A Slovníček pojmů pro bezpečnostní protokoly **58**

B Verze Linux jádra dle verzí systému Android **60**

Kapitola 1

Úvod

Globální problém s mobilními aplikacemi, které k poskytování svých služeb využívají internetové připojení je, že žádná z nich neposkytuje informace o využívaných protokolech. Zejména uživatelé připojující se k veřejným sítím jsou ohroženi, protože útočníci mohou odposlouchávat nedostatečně zabezpečenou komunikaci. Díky znalosti využitého komunikačního protokolu a dat poskytnutých aplikaci může zkušený uživatel rozhodnout, zda je aplikace důvěryhodná nebo ne.

Tato diplomová práce si klade za cíl implementovat Android aplikaci, pojmenovanou AppCheck, která umožňuje monitorování míry bezpečnosti komunikace ostatních aplikací. Čtenář získá informace o tom, jak obejít omezení systému Android při získávání informací z jiných aplikací včetně popisu řešení hlavních problémů analýzy jejich síťové komunikace. Konkrétněji se práce zaměřuje na zpracování paketů uvnitř mobilního zařízení a získávání datového toku jednotlivých aplikací. Díky řešení těchto problémů lze určit, nakolik je monitorovaná aplikace bezpečná. AppCheck dokáže zachytit síťový tok jiné aplikace a uložit ji do souboru pcap, který je čitelný např. pomocí nástroje Wireshark. Tento soubor slouží k analytickému rozpoznání, kolik paketů bylo zasláno bezpečným způsobem. Všechny tyto funkce jsou poskytovány bez nutnosti vyšších oprávnění, než má běžná aplikace.

V teoretické části jsou popsány potřebné principy a souvislosti poskytující ucelený přehled týkající se potencionálního rizika, které může mít za následek zneužití osobních údajů uživatele. Poskytuje informace o bezdrátových sítích, včetně popisu protokolů zajišťujících důvěrnost komunikace. Navazující část se zaměřuje na návrh a implementaci aplikace. Zároveň popisuje problémy, které bylo nutné vyřešit pro její realizaci. Jsou zde zmíněny i jednotlivé experimenty, předcházející samotný návrh. Ty měly za cíl zjistit, zda jsou požadované funkcionality technicky proveditelné. V závěru práce jsou shrnuty dosažené cíle včetně experimentů prováděných na již funkční aplikaci.

Kapitola 2

Mobilní zařízení a jejich bezpečnostní rizika

První část kapitoly je zaměřena na popis bezpečnostních mechanismů, jež se využívají k zabezpečení komunikace uživatele v bezdrátových sítích. Jedná se o protokoly a algoritmy, které nejsou závislé na konkrétním operačním systému. Je zde soustředěna pozornost zejména na popis protokolů patřících do skupiny IEEE 802.11. Ostatní bezdrátové sítě jakými jsou GSM, UMTS, Bluetooth a další, nejsou pro tuto práci podstatné, proto zde nebudou dále zmiňovány. Při bližším zájmu o zabezpečovací mechanismy těchto sítí, lze nahlédnout například do knihy paní Pužmanové [15], jenž se danému tématu věnuje.

Oproti tomu protokoly pracující na úrovni aplikace, popsané v kapitole 2.2, jsou pro tuto práci stěžejní. Jedná se o protokoly aplikační vrstvy referenčního síťového modelu. Jsou využívány nejen v mobilních zařízeních, ale i v ostatních aplikacích komunikujících skrze internetové připojení. Dále jsou zmíněna rizika, která se týkají senzorů a přístupových technologií sloužících k autentizaci uživatelů. Společně s využitím přístupu k Internetu mohou vést ke ztrátě soukromí. V závěru kapitoly je popsán operační systém Android, především z pohledu jeho implementace, která je založená na jádře operačního systému Linux.

2.1 Bezdrátové sítě a jejich bezpečnostní protokoly

Myšlenka bezdrátových síti je s námi již od 60. let minulého století. Prvním úspěšným pokusem o zprovoznění bezdrátové sítě se může pochlubit tým z University of Hawaii[1], kterému se podařilo zprovoznit síť s názvem *The Aloha System*. Článek *THE ALOHA SYSTEM — Another alternative for computer communications* [1] popisuje implementaci této sítě. Jejím hlavním rysem bylo, že fungovala jako radiová síť. Tzn. informace přes ni poslané se šířily pomocí radiových vln. Tento úspěch byl zajisté historickým zlomem, i když její rychlost byla pouze 24000 baudů. Nicméně otázka standardizace se začala řešit až kolem roku 1991, tj. přibližně o 20 let později. Po několika letech domluv byl v roce 1997 publikován standard 802.11[2] (s běžným označením Wi-Fi), který zastřešoval dosavadní bezdrátové sítě. Byl vytvořen společností IEEE[3], která je jednou z předních světových společností pro tvorbu standardů v oblasti elektroniky, elektrotechniky a informačních technologií.

[1] Stránky univerzity: https://www.hawaii.edu/
[2] RFC pro 802.11: https://tools.ietf.org/html/rfc7494
[3] Celým názvem *Institute of Electrical and Electronics Engineers*. Viz https://www.ieee.org/

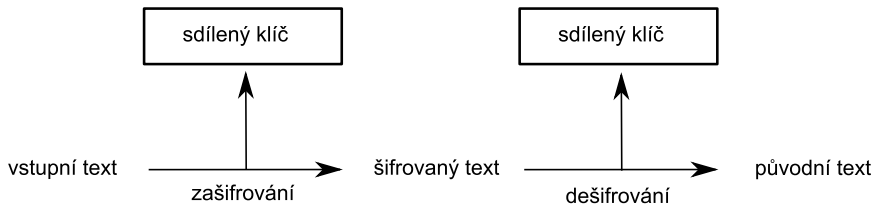
Od konce 90. let minulého století se standard 802.11 stále vyvíjí. Vzniklo již sedm derivací IEEE 802.11, které se od sebe liší koncovým písmenným označením. Rozlišujeme verze: a, b, g, n, y, ac, ad. Nejnovějším z této rodiny se má stát IEEE 802.11ax, který by měl spatřit světlo světa v roce 2019. Standardy neobsahují jen popis samotného přenosu dat. Jejich neméně důležitou součástí jsou i specifikace bezpečnosti přenosu. Existují tři bezpečnostní protokoly: jmenovitě WEP, WPA a WPA2. Pravděpodobně k nim časem přibudou i další, přičemž v roce 2019 by měl být publikován protokol WPA3.

Zmíněnými protokoly se bude zabývat jen teoretická část této práce. Důvodem je, že se navržená aplikace nezaměřuje na analýzu bezpečnosti bezdrátových sítí. Jejím cílem je analyzovat ostatní nainstalované aplikace a ty nemohou ovlivnit, jakým způsobem se mobilní zařízení k Internetu připojuje. To ovlivňuje uživatel a proto je vhodné zmínit, jaké jsou vlastnosti poskytovaných zabezpečovacích protokolů. K této kapitole je sepsán slovníček pojmů, jenž se nachází v příloze A a do kterého je vhodné nahlédnout, pokud čtenář narazí na jemu neznámé pojmy.

2.1.1 Wired Equivalent Privacy (WEP)

Wired Equivalent Privacy (volně přeloženo jako „soukromí srovnatelné s drátovým přenosem“) je prvním bezpečnostním protokolem pro bezdrátové sítě. Byl publikován v roce 1999 jako součást standardu 802.11, avšak je nepovinným aspektem sítě. Jak již název napovídá, jeho cílem je, aby se bezdrátový přenos stal stejně bezpečným, jako drátový. Implementuje několik mechanismů, které si kladou za cíl zvýšit bezpečnost komunikace.

Prvním z mechanismů zabezpečení je kontrola integrity zpráv. V protokolu WEP je řešena pomocí 32 bitového cyklického redundantního součtu (zkratka CRC má v angličtině dva možné významy: *Cyclic redundancy checks* nebo *Cyclic redundancy codes*). Informace týkající se CRC32 byly čerpány ze článku *32-bit cyclic redundancy codes for Internet applications* [10], jehož autorem je pan Koopman, který se v něm dále zabývá možnými vylepšeními výkonu těchto kódů.

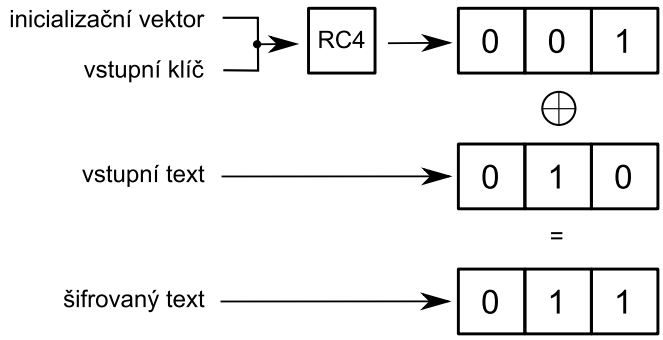


Obrázek 2.1: Princip symetrické kryptografie. Vstupní text je šifrován klíčem, který je sdílený pro obě komunikující strany. Přes síť se data přenáší zašifrovaně. Na straně příjemce jsou dešifrována stejným klíčem na původní text.

Dalším mechanismem je šifrování dat. Je zde implementována symetrická šifra (princip symetrické šifry je naznačen na obrázku 2.1), konkrétně proudová šifra RC4 vytvořená panem Ronaldem Rivestem z firmy RSA Data Security[4] v roce 1987. Bezpečnost samotné šifry spočívala zejména v utajení její implementace. To se změnilo v září roku 1994, kdy byl kód zveřejněn skrze anonymní e-mail skupině Cypherpunks[5]. Následně byla šifra rozluštěna

[4] V současnosti ji najdeme pod názvem RSA Security.
[5] Původní e-mail viz http://cypherpunks.venona.com/date/1994/09/msg00304.html

během několika dní panem Bobem Jenkinsem[6]. Po zveřejnění došlo k porušení obchodního tajemství a tento algoritmus získal mnoho nových označení, jako jsou ARC4 nebo ARCFOUR. I přes všechna úskalí měla RC4 stále značné využití a stala se součástí mnoha protokolů i standardů pro bezdrátovou komunikaci. Byla oblíbená díky své rychlosti a jednoduchosti implementace. Pro bližší představu, jak je do šifrování zprávy zapojena RC4, poslouží obrázek 2.2. Vstupem pro RC4 je inicializační vektor a vstupní klíč. Vzniklé bitové pole se následně pomocí funkce XOR spojí se vstupním textem a vytvoří tak kryptogram. Dešifrování má stejný proces. Jedinou změnou je, že se převádí šifrovaný text na původní zprávu.



Obrázek 2.2: Ukázka šifrování pomocí proudové šifry RC4. Kryptogram čili šifrovaný text, vzniká pomocí bitové funkce XOR, jenž je aplikována na šifrovací proud a vstupní text. Dešifrování probíhá stejně, jen místo kryptogramu se výpočtem funkce XOR získá původní text.

Poslední bezpečnostní mechanismus je kontrola přístupu do sítě, čili autentizace. WEP poskytuje dva typy – otevřený systém (*open system*) a jednostrannou autentizaci se sdíleným klíčem (*shared key*). Otevřená autentizace je u tohoto protokolu implicitní a probíhá pomocí SSID[7], tj. identifikace klienta. Klienti, kteří se chtějí připojit k bezdrátové síti, pošlou své SSID přístupovému bodu, jenž jim následně umožní přístup. U jednostranné autentizace se sdíleným klíčem je využita symetrická kryptografie. Délka klíče, se kterým se u WEP můžeme setkat, je 40b, popřípadě 104b u vylepšené verze protokolu. Ke klíči se pak připojuje 24b, jež slouží jako inicializační vektor. Výsledná velikost sdíleného klíče proto může být 64b, případně 128b.

Problémem tohoto protokolu je zejména fakt, že implementuje slabé autentizační mechanismy a nepodporuje obousměrnou autentizaci. Vynucuje pouze to, aby se uživatel autentizoval síti, nikoliv i síť uživateli. To s sebou nese značné riziko, protože uživatel nedokáže poznat, zda není síť podvržená. CRC32[8], které je zde použito pro zajištění integrity dat, nechává prostor pro útok *man in the middle* (blíže popsán v příloze A), díky kterému je útočník schopen zachytit veškerou komunikaci. Pro přesnější popis útoku a jeho ukázku lze nahlédnout do kapitoly *0x451 Man-in-the-Middle Attacks* v knize pana Ericksona [3]. I šifrování protokolu pomocí RC4 je do jisté míry zranitelné, což blíže popisuje článek *Weaknesses in the Key Scheduling Algorithm of RC4* [4].

Při zabezpečení bezdrátových sítí by se WEP již neměl používat. Nevhodný je od roku 2001, kdy došlo k prvnímu úspěšnému útoku, který se označuje jako *FSM attack*. Název

[6] Viz vlákno komunikace k původnímu e-mailu zveřejňujícímu kód RC4: https://groups.google.com/forum/#!msg/sci.crypt/JsO3xEATGFA/-wO4ttv7BCYJ

[7] *Service Set Identifier* je identifikátor zařízení v bezdrátové síti.

[8] *Cyclic Redundancy Code* s délkou 32 bitů.

je odvozený od kolektivu autorů: Fluhrer, Mantin a Shamir. Těm se podařilo pouhým odposloucháváním zašifrované komunikace získat vždy několik prvních bitů v paketu. A to jen díky tomu, že začátky přenášených paketů v nešifrovaném textu jsou snadno předvídatelné. Navíc inicializační vektory jsou spolu s pakety posílány nešifrovaně, což útočníkovi pomáhá zjistit, jak vypadají první tři bajty paketu. Tímto útokem dokázali autoři získat celý vstupní klíč pro šifru RC4. Nejrychlejší získání vstupního klíče z WEP bylo uskutečněno za méně než minutu. Povedlo se to autorům článku *Breaking 104 Bit WEP in Less Than 60 Seconds* [18]. Těm se podařilo prolomit 104b vstupní klíč za čas mezi 54 a 58 sekundami, přičemž pravděpodobnost úspěchu vyčíslili na 50%. Další informace ohledně útoků na protokol WEP jsou přehledně sepsány v knize *Hacking* [3] v kapitolách *0x480 WEP Attacks* až *0x485 Fluher, Mantin, and Shamir (FMS) Attack*. Nejznámnější WEP útok, tj. FMS útok, je zde popsán velmi dopodrobna včetně kódů a dovysvětlení.

2.1.2 Wi-Fi Protected Access (WPA)

Wi-Fi Protected Access[9] (volně přeloženo jako *zabezpečený přístup k Wi-Fi*) je následníkem bezpečnostního protokolu WEP. Byl vytvořen firmou Wi-Fi Alliance[10], která se zaměřuje na certifikaci HW a SW produktů pro využití ve Wi-Fi sítích. Tato firma vlastní ochrannou známku Wi-Fi, kterou by měly používat pouze produkty, jenž byly touto společností otestovány a zaručují tak dostatečnou míru kvality.

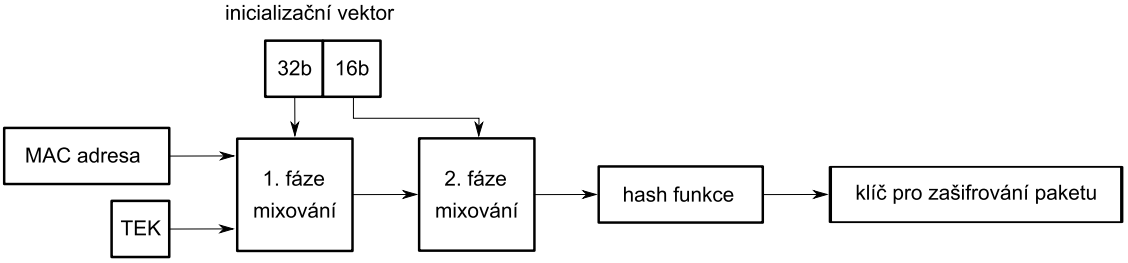
Protokol se začal rýsovat v roce 2002 a standardizován byl v roce 2004 společností IEEE. Vznikal poměrně narychlo, aby co nejdříve pokryl nedostatky WEPu. Původně měl implementovat veškeré náležitosti standardu 802.11i, avšak potřeba rychlé náhrady stávajícího protokolu tento plán změnila a výsledné WPA implementuje pouze jeho část. Využívá šifru RC4, stejně jako protokol WEP, aby byla zajištěna zpětná kompatibilita. Běžně používá 128b klíč a 48b inicializační vektor.

Integrita zpráv je zajištěna pomocí stávajícího algoritmu CRC32, který je využíván i protokolem WEP, a nově přidaného algoritmu *Message Integrity Code*, dále jen MIC. Principem tohoto algoritmu je přidání digitálního podpisu ke každé zprávě. Ten je vypočítán pomocí algoritmu *Michael* na základě pořadového čísla paketu, zdrojové a cílové MAC adresy, datové části rámce a náhodné hodnoty. Pro bližší představu, jak algoritmus *Michael* funguje a jaké jsou jeho slabiny lze nahlédnout do článku *Security Analysis of Michael: The IEEE 802.11i Message Integrity Code* [6]. O algoritmu platí, že je bezpečnější než CRC32, protože umí zabránit *man in the middle* útoku. Pro šifrování dat se i nadále využívá šifra RC4, která je podpořena protokolem *Temporal Key Integrity Protocol* (TKIP). Ten specifikuje tři bezpečnostní mechanismy. Kontrolu integrity zpráv, která chrání před *man in the middle* útokem. Číslování paketů, které slouží jako ochrana před *replay attack*. A dynamickou správu klíčů, která zajišťuje využívání různých klíčů v různém čase.

U WPA se pro každý paket vytváří nový klíč určený k jeho zašifrování. Schéma tvorby klíče je na obrázku 2.3. Vstupem je 32b z inicializačního vektoru, MAC adresa a *Temporal Encryption Key* (TEK), který slouží pro šifrování dat a zabezpečení provozu v unicastové komunikaci. Jejich vzájemnou kombinací se vytvoří vstup 2. fáze, ke kterému se přimixuje zbylých 16b inicializačního vektoru. Aplikací hash funkce na výsledek 2. fáze se získá nový jedinečný 128b klíč, jímž bude zašifrován konkrétní odchozí paket.

WPA se dělí na dvě varianty: WPA-Personal a WPA-Enterprise. Jejich rozdíl spočívá v různosti implementace autentizačních mechanismů. WPA-Personal, označované také jako

[9] Informace o protokolu WPA byly čerpány zejména z knihy [15].
[10] Stránky společnosti Wi-Fi Alliance: https://www.wi-fi.org/



Obrázek 2.3: Mixování klíče pro zašifrování paketu se skládá ze dvou fází. V první se kombinuje MAC adresa, dočasný klíč TEK a 32b z inicializačního vektoru. Druhá fáze vytváří klíč z výstupu první fáze a zbylých 16b inicializačního vektoru. Průchodem hash funkcí vzniká výsledný šifrovací klíč.

WPA-PSK, používá pro autentizaci předsdílený klíč (*Pre-Shared Key*, zkráceně PSK). Je to stejný typ klíče, jaký je používaný u protokolu WEP. U WPA-PSK platí, že k jednomu ESSID[11] náleží právě jeden PSK, nicméně neslouží k šifrování celé zprávy, jako tomu je v případě protokolu WEP. V případě WPA-PSK se používá pro generování párových klíčů *Pairwise Transient Key*, zkráceně PTK, potřebných pro unicastovou komunikaci. Právě díky využití předsdíleného klíče poskytuje tato varianta, menší míru zabezpečení než WPA-Enterprise, avšak je uživatelsky dostupnější a využívanější díky jednoduššímu nasazení v provozu.

WPA je stále využívaným protokolem a při správné implementaci je dostatečně bezpečný. Eliminovaly se útoky jako je *repeat attack*, *man in the middle attack*, *FMS attack* a další, spojené s nedostatečným inicializačním vektorem či slabým klíčem. Nicméně některé z problémů zde zůstaly a to především kvůli spěchu při vývoji. Jedním z nich jsou slabé klíče v případě využití WPA-PSK. Ty se dají uhádnout například použitím slovníkového útoku. Při něm se automatizovaně zkouší jedno heslo za druhým pomocí předpřipraveného slovníku. Na tuto slabinu poukázal jako první pan Moskowitz ve svém článku *Weakness in Passphrase Choice in WPA Interface* [14]. Tento útok lze provést automatizovanými nástroji, jako je Aircrack-ng[12].

2.1.3 Wi-Fi Protected Access 2 (WPA2)

WPA2[13] se považuje za doposud nejlepší řešení zabezpečení v bezdrátových sítích. Je následovníkem protokolu WPA (viz předchozí kapitola). Oba vychází ze stejného standardu IEEE 802.11i. Protokol WPA implementoval tento standard jen z části, oproti tomu WPA2 ho implementuje celý. Díky tomu se označení WPA2 a 802.11i v mnoha článcích ztotožňují.

Upouští se od zavedených mechanismů autentizace a šifrování. Avšak kvůli zpětné kompatibilitě je možné implementovat šifrování dat pomocí RC4 a protokolu TKIP. Proudová šifra RC4 je v protokolu WPA2 nahrazena blokovou šifrou *Advanced Encryption Standard*, zkráceně AES, v módu *Counter Mode CBC-MAC Protocol*, zkráceně CCMP. Jedním společným rysem obou šifer je, že jsou symetrické. Pro autentizaci využívá WPA2 primárně protokol *Counter Mode CBC-MAC Protocol* (dále jen CCMP). *Cipher Block Chaining* čili CBC je jedním z režimů blokových šifer (*Block Encryption Modes*). Jeho využitím se před-

[11] *Extended Service Set Identifier* je identifikátor označující název jedné konkrétní bezdrátové sítě.
[12] Dostupný na stránkách: https://www.aircrack-ng.org/
[13] Informace ohledně protokolu byly převzaty z článku [11].

chází problému s opakujícími se kryptogramy[14], protože zde dochází k náhodnému mixování bloků vstupního textu a následnému výpočtu binární funkce XOR s předcházejícím zašifrovaným blokem. Pro správné dešifrování zpráv je pak nutné, aby byla použita inverzní bloková šifra následovaná výpočtem binární funkce XOR ve správném pořadí.

V knize od autorů Meyera a Matyase [5] je CBC-MAC označeno jako nejrozšířenější způsob použití blokové šifry pro kryptografický kontrolní součet (dále jen MAC). Jedná se o variantu, kde je vypočítaný MAC připojen za bloky původní zprávy a je s ním zacházeno, jako by byl její součástí. To znamená, že blok s MAC je šifrovaný společně s ostatními. Díky tomuto přístupu se při jeho dešifrování dá uplatnit stejný postup, jako u ostatních bloků. Pro následnou kontrolu, zda nebyla porušena integrita, stačí porovnat dešifrovaný a na straně příjemce vypočítaný MAC. Zpráva byla přenesena v pořádku, pokud se obě hodnoty shodují.

Jedním z rozdílů protokolů WPA a WPA2 jsou mechanismy použité pro šifrování. WPA využívá pro šifrování protokol TKIP a šifru RC4, kdežto WPA2 používá protokol AES-CCMP. To, v čem jsou si podobné je, že oba implementují dva režimy, které se od sebe liší implementací autentizace. Prvním je WPA2-Personal pro osobní použití a druhým WPA2-Enterprise pro využití ve firmách. Jednotlivé typy režimů jsou u obou protokolů velmi podobné. WPA2-Personal, stejně jako WPA-Personal, používá pro autentizaci klíč PSK. WPA2-Enterprise a WPA-Enterprise využívají pro autentizaci protokol 802.1X.

Stále platí, že WPA2 je nejbezpečnějším z dostupných bezpečnostních protokolů pro bezdrátové sítě. Avšak i on má své slabiny. U varianty WPA2-Personal přetrvává problém s předsdíleným klíčem, který se vyskytuje u WEP i WPA-Personal. Nejznámějším útokem na WPA2, je *Key Reinstallation Attack* neboli KRACK[15]. První úspěšný KRACK útok byl uskutečněn v roce 2017 panem Vanhoefem z výskumné skupiny imec-DistriNet[16]. Útok je cílený přímo na slabinu implementace tohoto protokolu. Konkrétně na *four-way handshake*, kdy dochází k ustanovení komunikace mezi klientem a přístupovým bodem. Během tohoto procesu si mezi sebou vyměňují i hlavní párový klíč (PMK), ze kterého se odvozují další klíče potřebné k šifrování. KRACK využívá zpráv, které byly při *four-way handshake* zaslány již dříve. Zaměřuje se na chybu, kdy implementace protokolu dovoluje znovu zaslat zprávy při výměně jednorázového klíče v případě, že druhá komunikující strana nereaguje správně. Běžně by to znamenalo, že je komunikace jen dočasně přerušena a proto se přístupový bod snaží o opětovné doručení zprávy. Ze strany přístupového bodu pak lze poslat několik zpráv, které mají za následek resetování čítačů. Díky tomu dokáže útočník podstrčit klientovi jím preferované hodnoty a tím oslabit samotné šifrování. Získá tak údaje potřebné pro výpočet šifry a může se pokusit odhadnout použitý klíč na základě známé části komunikace. Po získání použitého klíče může dešifrovat všechny zprávy v rámci této komunikace.

[14] Kryptogram – zašifrovaný vstupní text

[15] Informace byly převzaty z článku: https://www.root.cz/clanky/sifrovani-wpa2-bylo-prolomeno-wi-fi-site-je-mozne-odposlouchavat/

[16] Stánka skupiny imec-DistriNet: https://distrinet.cs.kuleuven.be/

2.1.4 Wi-Fi Protected Access 3 (WPA3)

Budoucností zabezpečení bezdrátového přenosu by se měl stát protokol WPA3[17]. Měl by přinést zjednodušení konfigurace Wi-Fi sítí a čtyři nové funkce. První z nich by měla poskytnout robustní zabezpečení uživatelům, kteří si volí slabá či nevyhovující hesla. Spolu s tímto bodem souvisí i snaha o posílení soukromí ve veřejných sítích pomocí individuálního šifrování dat. Dále se vývojáři zaměří na pohodlnější proces konfigurace zařízení, která nemají uživatelské rozhraní, nebo ho mají značně limitované. V neposlední řadě chtějí poskytnout 192-bitovou bezpečnostní sadu spolu s podporou CNSA[18] pro sítě s potřebou značných bezpečnostních opatření. Zároveň společnost Wi-Fi Alliance potvrdila, že protokol WPA2 bude i nadále podporován a napnou maximální úsilí, aby nezastarával a stále dostačoval pro běžné použití.

2.2 Zabezpečení komunikace na úrovni aplikací

Při vývoji bezpečných aplikací je vhodné se nespoléhat jen na zabezpečení nižších vrstev síťového modelu, jako je L2 či L3 vrstva. Tvůrce aplikace nedokáže ovlivnit implementaci bezpečnostních opatření v těchto vrstvách, proto se využívají podpůrné mechanismy zabezpečení na úrovni aplikací, které se již stávají požadovaným standardem. Jako příklad poslouží webový prohlížeč Chrome od firmy Google. Během roku 2018 oznámila firma rozhodnutí již dále nezobrazovat ikonku zeleného zámečku, jenž měla za cíl uvědomit uživatele, že je daná stránka zabezpečená. Nová politika Google počítá s tím, že všechny stránky by měly být zabezpečené. Chtějí tedy naopak upozorňovat jen na ty stránky, které zabezpečené nebudou, tzn. stránky které nebudou implementovat protokol HTTPS.

Tato kapitola je věnována možným zabezpečením aplikací. Podrobněji je zde popsán aplikační protokol HTTP sloužící pro získávání dat, včetně navazujícího HTTPS, který krom funkcionalit shodných s HTTP řeší i otázku bezpečnosti. Dále zde budou přiblíženy technologie SSL a TLS. Ty zajišťují zabezpečení na úrovni aplikace, proto jsou vhodným doplňkem ke zmíněným aplikačním protokolům.

2.2.1 Hypertext Transfer Protocol (HTTP)

HTTP[19] je aplikační protokol pracující na 7. vrstvě síťového modelu. Jeho koncept a implementaci vytvořil pan Tim Berners-Lee v roce 1989. O rok později jej využil pro systém *World Wide Web*[20], zkráceně WWW. Funguje na principu zasílání zpráv a modelu klient-server. Když u HTTP zmiňujeme klienta, tak máme typicky na mysli webový prohlížeč. Jeho komunikačním protějškem bývá webový server. Existuje několik verzí tohoto protokolu. Nejdéle používanou verzí je HTTP/1.1, která je aktivně využívána přibližně 20 let.

[17] Dle článku: https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-security-enhancements

[18] *Commercial National Security Algorithms*, volně přeloženo jako Komunikační národní bezpečnostní algoritmus. Viz stránka: https://tools.ietf.org/id/draft-jenkins-cnsa-cert-crl-profile-00.html

[19] Informace týkající se protokolu HTTP byly čerpány z knihy *Network protocols handbook* [21] a RFC 2616 https://tools.ietf.org/html/rfc2616.

[20] *World Wide Web* je systém, který umožňuje sdílení dokumentů v rámci Internetu. Bližší popis lze nalézt na stránkách w3 organizace, kterou založil vynálezce systému WWW: https://www.w3.org/People/Berners-Lee/1996/ppf.html

Pro adresaci zdroje dat se využívá *Uniform Resource Identifier*[21] (URI). Speciálními případy URI jsou *Uniform Resource Locator*[22] (URL) a *Uniform Resource Name*[23] (URN). HTTP implementuje zprávy typu dotaz-odpověď (anglicky *Request-Response*), pomocí nichž mezi sebou komunikují klienti a servery. Z hlediska jednotlivých částí zprávy se tyto dva typy příliš neliší. Obě hlavičky jsou zasílány jako čitelný text. V prvním řádku, označeném jako *request line*, má zpráva pro dotaz definovaný požadavek pomocí označení konkrétní metody, jakou je například GET, POST, CONNECT a další. Oproti tomu zpráva s odpovědí obsahuje ve stejném místě stavový kód (*status line*) a jeho zdůvodnění. To udává, zda byl dotaz odpovězen správně, například kód 200 OK a podává dovysvětlení v případě, že došlo k chybě, například kód 400 Bad request. Stavových kódů je definováno více než je zde zmíněno a každý z nich blíže určuje, jak se server s dotazem vypořádal. Za touto částí následuje u obou zpráv *general header*. Na ni navazuje hlavička, která je závislá na typu zprávy, tj. *request header* nebo *response header*. Poslední dvě části, *entity header* a *message body*, jsou u obou opět stejné.

Pro bližší představu, jak mohou oba typy zprávy vypadat, slouží ukázka 2.1, jakožto zástupce dotazu na server. A ukázka 2.2, která obsahuje odpověď od serveru. V obou výpisech se dá najít v prvním řádku komunikační protokol. V tomto případě to je HTTP/1.1. Jedná se o úspěšný dotaz na server vyvolaný aplikací Liftago[24]. O úspěšnosti provedeného dotazu svědčí stavový kód v odpovědi, tj. 200 OK. Komunikace byla zachycena při otevření internetové stránky skrze rozhraní internetového prohlížeče v rámci nativní aplikace. Tyto stránky jsou v Android aplikacích označovány jako *WebView*[25].

```
GET /mobile/trackpoint/?pm=342521&ADFPageName=Liftago+Taxi%7CAndroid%7C-
Start&ADFdivider=|&md=ChBNT0JJTEVfQVBQX1NUQVJUEnQKJDA3ZGUxZWY2LWE3NmYtN-
DI0Yy1hMzVmLWQyNTk2M2NjZDAz%0AYSIkY29tLmFkbGVyaXRlY2guYXBwLmxpZnRhZ28uc-
GFzc2VuZ2VyKg HTTP/1.1
Host: track.adform.net
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Linux; Android 6.0; Classic_LTE Build/MRA58K; wv)
AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/69.0.3497.100
Mobile Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: cs-CZ,en-US;q=0.9
X-Requested-With: com.adleritech.app.liftago.passenger
```

Výpis 2.1: Ukázka hlavičky dotazu na internetový server track.adform.net pomocí metody GET protokolu HTTP verze 1.1. Z odeslané hlavičky lze zjistit, že byla zpráva poslána ze zařízení Classic_LTE, které má operační systém Android 6.0. Podle názvu balíčku aplikace, která vyvolala tento dotaz lze soudit, že se jedná o aplikaci Liftago.

[21] URI je formát, který specifikuje konkrétní zdroj na základě jeho identifikátoru.

[22] URL definuje zdroj na základě jeho umístění a mechanismu, jakým se dá získat. Běžně se s ním můžeme setkat v souvislosti s protokolem HTTP, ale není to jediný protokol, který URL využívá.

[23] URN specifikuje zdroj na základě jeho jména, nikoliv však umístění.

[24] Liftago: https://play.google.com/store/apps/details?id=com.adleritech.app.liftago.passenger

[25] Dokumentace pro WebView: https://developer.android.com/reference/android/webkit/WebView

```
HTTP/1.1 200 OK
Server: nginx
Date: Fri, 12 Oct 2018 09:08:09 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Keep-Alive: timeout=15
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Encoding: gzip
Expires: -1
Vary: Accept-Encoding
Access-Control-Allow-Origin: *
P3P: CP="NOI DSP COR NID CURa ADMa DEVa TAIa PSAa PSDa OUR LEG NAV INT"
```

Výpis 2.2: Ukázka hlavičky odpovědi na dotaz pomocí metody GET, jehož hlavička je uvedena ve výpisu 2.1. Server vrátil požadované informace pomocí protokolu HTTP/1.1, kde 1.1 udává verzi protokolu. Data byla navrácena v pořáku, o čemž svědčí stavový kód 200 OK.

Problém HTTP protokolu spočívá zejména v tom, že veškerá komunikace přes něj přenesená je nezabezpečená, tzn. přenáší se bez dodatečných mechanismů pro zabezpečení osobních dat. I v RFC 2616 je v kapitole *15.1.3 Encoding Sensitive Information in URI's* zmíněno, že by se s pomocí HTTP neměly zasílat citlivá data. Poměrně často se při práci s mobilními aplikacemi setkáváme s dotazem pomocí metody GET. Využívá se pro získávání dat z URI adresy, na kterou se konkrétní dotaz zasílá. Využívá se také pro získávání dat týkajících se internetových stránek, díky kterým mohou být následně zobrazeny uživateli.

Tato metoda je kritická ve chvíli, kdy se v hlavičce pošlou nezašifrované tajné informace. Mohou to být například hesla, přístupové klíče, osobní údaje, atp. Čitelnost daných URI je problematická i tehdy, pokud nejde přímo o útok na zasílaná data v reálném čase. Mobilní aplikace mohou ukládat informace o chování uživatelů za účelem následné analýzy, k čemuž se většinou využívají nástroje třetích stran. Mezi taková data mohou spadat i údaje o dotazovaných URI. Díky tomu jsou pak nešifrovaně zaslané informace získatelné z více zdrojů a potencionálně méně v bezpečí. V případě zasílaní citlivých údajů v požadavku na server je proto vhodnější metoda POST. Ta je podobná metodě GET, avšak narozdíl od ní lze přenášet ciltivé údaje v těle, přičemž u GET to lze pouze jako součást URI. Více informací ke specifikaci protokolu HTTP lze nalézt v RFC 2616[26].

2.2.2 Hypertext Transfer Protocol Secure (HTTPS)

Protokol HTTPS[27], jiným označením HTTP Over TLS, je vybudován na základě protokolu HTTP a slouží ke zvýšení bezpečnosti spojení mezi webovým prohlížečem a webovým serverem. Díky tomu je stejně jako HTTP klient-server protokolem. Důvodem jeho vzniku bylo nulové zabezpečení v protokolu HTTP. HTTPS využívá dodatečné bezpečnostní me-

[26] Stránky odkazující na RFC 2616: https://tools.ietf.org/html/rfc2616
[27] Specifikaci protokolu HTTPS lze nalézt v RFC 2818 https://tools.ietf.org/html/rfc2818

chanismy, jako *Secure Socket Layer* (SSL) nebo *Transport Layer Security Layer* (TLS) pro zabezpečení přenášených dat.

HTTPS používá stejné zprávy, tj. *Request* a *Response*, jako HTTP pro komunikaci mezi klientem a serverem. Rozlišit tyto protokoly v zaznamenané síťové komunikaci lze například podle čísla využívaného portu. Zatímco HTTP využívá primárně TCP portu 80 pro komunikaci přes nižší vrstvy TCP/IP, tak HTTPS využívá portu s číslem 443.

2.2.3 Secure Sockets Layer (SSL)

Pomocí SSL lze zabezpečit jednotlivé aplikace, nikoliv však celou komunikaci. Jedná se konkrétně o autentizaci, šifrování dat a zajištění jejich integrity. Využívá se při implementaci protokolu HTTPS, jak již bylo zmíněno v podkapitole 2.2.2 a pracuje nad transportním protokolem *Transmission Control Protocol* (TCP). Pro zašifrování dat se používají blokové šifry v režimu CBC.

SSL poskytuje několik možností implementace autentizace, včetně obousměrné autentizace. U SSL lze využít jmen a hesel v případě použití serveru RADIUS, digitálních certifikátů X.509 či jmen a tokenů, tj. RSA Securid. Do jisté míry můžou být některé implementace náchylné na útok *man in the middle*. Zvláště v případě, když jsou používány certifikáty X.509. Při autentizaci si SSL klient získává certifikát od serveru, ke kterému se chce připojit. Je-li certifikát podvržený, tak zobrazí uživateli informaci o případných problémech, jenž u certifikátu odhalil. Pokud nebere uživatel tuto výstrahu na zřetel a udělí certifikátu výjimku, tak se snadno stane obětí útoku.

V roce 2014 byl uskutečněn útok *Padding Oracle On Downgraded Legacy Encryption*, zkráceně POODLE[28], jenž cílil na SSLv3[29]. Využíval slabiny algoritmu CBC. Ten vytváří pro zašifrování každého bloku nový inicializační vektor, který vzniká závisle na bloku předcházejícím. Díky tomu jsou všechny inicializační vektory jistým způsobem propojené. Tento útok však nelze použít na TLS, což je způsobeno tím, že při šifrování generuje nový inicializační vektor pro každou zprávu. Avšak lze vynutit, aby se síť zabezpečená pomocí TLS přepnula na nižší verzi, tj. SSLv3, kterou může mít volitelně implementovanou v rámci zachování zpětné kompatibility. To ve výsledku způsobí, že i sítě zabezpečené pomocí TLS jsou v režimu zpětné kompatibility zranitelné.

2.2.4 Transport Layer Security (TLS)

TLS je technologie zabezpečení komunikace, která vznikla jako odpověď na bezpečnostní problémy SSLv3, ze kterého vychází. Tvoří mezivrstvu mezi TCP/IP protokolem a protokolem jako je HTTPS nebo IMAP. Stejně jako SSL, i tato technologie zajišťuje soukromí, integritu a autentizaci mezi dvěma komunikujícími stranami. Dříve než se začne s přenosem dat je nutné, aby si mezi sebou klient a server ujednotili sadu algoritmů pomocí nichž bude komunikace zabezpečená. Ta se označuje jako *cipher suite* a je využívána i protokolem SSL. *Cipher suite* se skládá z několika kategorií algoritmů. V rámci jedné sady se definuje jeden algoritmus pro každou zmíněnou kategorii:

- *Key Exchange Algorithm* určuje, jakým způsobem se budou vyměňovat klíče pro šifrování komunikace.

[28] Útok POODLE je podobný útoku BEAST, který také využíval náchylnosti režimu CBC. Tento i další útoky jsou blíže popsány v článku *Security enhancements to TLS for improved national control* [2].

[29] Technologie zabezpečení SSL má jen dvě veřejné verze, tj. SSLv2 a SSLv3. Další vývojový stupeň protokolu byl pojmenován TLS.

- *Bulk Encryption Algorithm* definuje délku klíčů, typ šifry (bloková/proudová/AEAD) a délku inicializačních vektorů. Pokud je využívána bloková šifra, tak se určuje ještě velikost jednoho bloku.
- *Message Authentication Code Algorithm*, zkráceně MAC, slouží pro autentizaci zpráv.
- *Pseudorandom Function* generuje klíče ze sdíleného tajemství.

Do roku 2019 byly publikovány čtyři verze tohoto protokolu[30], přičemž každá podporuje rozdílnou sadu algoritmů. Nejnovější z nich, TLS 1.3, implementuje *cipher suite* naprosto odlišný od předchozích verzí a zakazuje značné množství starších algoritmů. Díky tomu nelze využít stejné sady pro TLS 1.3 a TLS 1.2 či starší. Dle pravidelného měsíčního průzkumu SSL Pulse[31], které vytváří firma Qualys Inc., se průběžně zvyšuje procento stránek s podporou TLS 1.3. Avšak TLS 1.2 stále zůstává nejpoužívanější.

2.2.5 X.509 a Certificate pinning

SSL certifikáty poskytují záruku důvěryhodnosti konkrétní služby. Standardem pro certifikáty je X.509[32]. Ve své podstatě je certifikát podpis veřejného klíče vygenerovaný pro konkrétní instituci či osobu, který je podepsaný certifikační autoritou[33], dále jen CA. Jednotlivé CA jsou vzájemně propojeny hierarchicky. Nejdůležitější z nich je kořenová CA, která podepisuje certifikáty dalším CA. Úspěšný útok na tento standard byl proveden již v roce 1993, kdy se podařilo vygenerovat z různých veřejných klíčů stejné certifikáty. To bylo možné, protože se pro generování klíčů využívala funkce MD5, u které byla v roce 2005 dokázána koliznost. Reálný útok cílený na MD5 provedla profesorka Wang Xiaoyun se svým týmem, přičemž podrobnější popis samotného útoku je uveden v jejím článku *How to Break MD5 and Other Hash Functions* [19]. Později se podařil obdobný útok i na některé další hash funkce.

Poněkud složitá hierarchie a menší důvěra v CA pramenící ze zdarma poskytovaných certifikátů od nedůvěryhodných CA byla impulsem pro hledání dalších možností, jak pomocí certifikátů zajistit důvěryhodnost. Obdobou se stalo tzv. *certificate pinning*, které nevyžaduje validaci certifikátu za běhu programu. Důvěryhodný certifikát je zahrnut již v nainstalované aplikaci a proto je toto řešení odolné vůči *man in the middle* útoku (ten je blíže popsán v příloze A). Tento přistup je značně využívaný při implementaci mobilních aplikací. Informace týkající se tohoto tématu byly převzaty z knihy *Security Intelligence: A Practitioner's Guide to Solving Enterprise Security Challenges* [12], konkrétně z kapitoly 1, podkapitoly *Certificate Pinning and OCSP Stapling*. Doposud nebyl proveden žádný úspěšný útok na tento typ ověřování důvěryhodnosti.

[30] Jednotlivé verze spolu s určením příslušného RFC: TLS 1.0 – RFC 2246; TLS 1.1 – RFC 4346; TLS 1.2 – RFC 5246; TLS 1.3 – RFC 8446. Jednotlivé dokumenty jsou dohledatelné na stránce https://tools.ietf.org.

[31] Jedná se o pravidelné měsíční monitorovaní kvality protokolů SSL a TLS na nejpoužívanějších stránkách. Dohledatelné na internetové stránce https://www.ssllabs.com/ssl-pulse/. Které stránky patří mezi nejpoužívanější jim poskytuje seznam populárních stránek projektu Alexa (https://www.alexa.com/). Jedná se o projekt firmy Amazon.com, který se mimo jiné zaměřuje na statistiku a analýzy toku dat na Internetu.

[32] Popsané v RFC 5280: https://tools.ietf.org/html/rfc5280

[33] Certifikační autorita je instituce, které důvěřuje jak majitel certifikátu, tak osoba či instituce, která podepsaný certifikát po jeho majiteli vyžaduje.

2.3 RESTful API

Pro získáváni dat ze serveru, se v aplikacích dá využít RESTful API. Je to aplikační programové rozhraní, které využívá architekturu REST (*Representational State Transfer*). Ta byla uveřejněna v roce 2000 v disertační práci pana Fieldinga [17], kde je popsána v kapitole 5. Architektura je značně provázána s protokolem HTTP, neboť pan Fielding je spoluautorem tohoto protokolu. Rozlišit, zda se jedná o REST architekturu, lze na základě splnění 6ti omezení:

- Je architektury klient-server – to je nezbytné, aby byly oddělena data od uživatelského rozhraní,
- Komunikace musí být bezstavová – každý požadavek musí obsahovat všechna data, která jsou podstatná pro získání relevantní odpovědi. Server si díky tomu nemusí uchovávat žádný stav,
- Má cache paměť – slouží pro zrychlení procesu, jsou-li některá data již přenesena a uložena v cache paměti, tak není nutné posílat znovu požadavek na stejná data. Eliminací počtu požadavků lze snížit latenci a zvýšit efektivitu poskytování dat,
- Jednotné rozhraní mezi komponentami – informace jsou mezi jednotlivými komponentami předávány v normalizované formě. Aby bylo možné se všemi komponentami komunikovat jednotně, tak bylo nutné zavést omezení rozhraní. REST definuje čtyři tyto omezení, konkrétně se jedná o: identifikaci zdrojů, manipulaci se zdroji pomocí reprezentace, samo-popisné zprávy a hypermedia,
- Vrstvené hierarchické uspořádání – architektura se skládá z oddělených vrstev, kde každá vrstva ví jen o svém bezprostředním okolí, tj. jen o vrstvách, se kterými může interagovat. To umožňuje lepší rozložení funkcionality a zlepšení škálovatelnosti systému a
- Podporuje styl „kód na vyžádání“ (*code-on-demand*) – nepovinné omezení, umožňuje rozšiřovat či upravovat funkčnost již nasazeného klienta za pomoci skriptů či appletů. Díky tomu redukuje požadavky na prvotní implementaci klienta.

Blíže jsou jednotlivá omezení popsána v kapitole 5.1 v již zmíněné disertační práci [17]. Vymezení pojmů týkajících se omezení rozhraní lze dohledat v kapitole 5.2 ve stejné práci.

Veškerá data, která jsou uložena na serveru mají vlastní URI, která slouží jako identifikátor. Pro práci s nimi slouží metody, které jsou využívány i ostatními technologiemi pro práci s databázemi. Jedná se o metody — *Create*, *Retrieve*, *Update* a *Delete* — souhrnně označované jako CRUD. Implementačně jsou řešeny za pomoci metod protokolu HTTP. Mapování REST metod na HTTP metody:

- Create~POST – vytvoří data na serveru,
- Retrieve~GET – získá požadovaná data ze serveru,
- Update~PUT – aktualizuje konkrétní data na serveru,
- Delete~DELETE – smaže data na serveru.

Mezi mobilními vývojáři je RESTful API oblíbenou součástí, zejména díky faktu, že poskytuje jednotné rozhraní pro komunikaci s komponentami a je snadno rozšiřitelné. Další výhodnou vlastností RESTful API je, že nezáleží na využívané platformě či implementačním jazyku. Díky tomu je snazší vyvíjet aplikaci na několika operačních systémech při využití toho stejného RESTful API.

2.4 Senzory a přístupové technologie

Jedním z bezpečnostních rizik značného procenta mobilních zařízení jsou senzory a přístupové technologie. Mezi standardní výbavu patří GPS, přední a zadní kamera, záznam zvuku a reproduktor. Dále se mohou objevovat přístupové technologie, jako je rozpoznání uživatele podle obličeje (například FaceID[34] od firmy Apple[35]), otisku prstu či jiných biometrických veličin.

Pokud si kupříkladu uživatel stáhne aplikaci na úpravu fotek, pak se dá očekávat, že bude vyžadovat přístup k fotoaparátu, aby s ním mohla pracovat. Uživatel si v takovém případě nemá důvod myslet, že by daná aplikace mohla být škodlivá. Potencionální hrozbou zde zůstává, že aplikace bude fotit, kdykoliv bude potřebovat, aniž by o tom uživatele informovala. Tato data pak může dále poskytovat jejímu provozovateli a schraňovat citlivé údaje o uživateli.

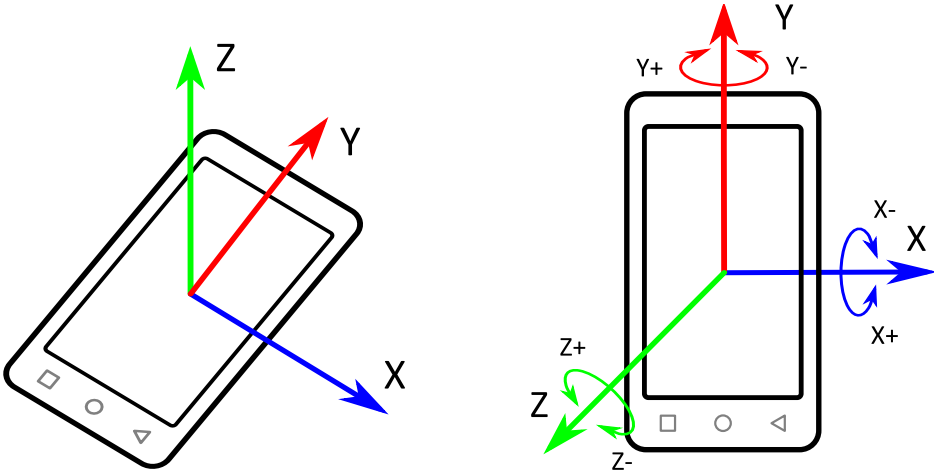
Nemusí se jednat jen o senzory zaznamenávající obraz či zvuk. Chytrá mobilní zařízení v sobě obsahují velké množství senzorů, u kterých nemusí být na první pohled jasné, jak by se daly zneužít. Na tento druh senzorů se zaměřili pánové Xu, Bai a Zhu. V roce 2012 publikovali článek *TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors* [20] týkající se aplikace TapLogger. Tu navrhli a implementovali, jako demonstraci možného útoku pomocí akcelerometru a gyroskopu, jejichž funcionalita je znázorněna na obrázku 2.4. Ze studie tohoto týmu navíc vyplývá, že data týkající se akcelerometru jsou dostupná jakékoliv nainstalované mobilní aplikaci bez nutnosti dodatečných oprávnění. Z takto získaných informací se jim podařilo zjistit, kterých míst displeje se uživatel dotýká. Poznatek pak uplatnili pro zjištění dat zadaných uživatelem. Jednalo se o informace, jako je PIN kód uživatele, jeho odemykací gesto nebo číslo bankovního účtu. Pro bližší informace o provedení útoku na tyto senzory je vhodné nahlédnout právě do zmiňovaného článku.

Potencionálně zranitelnou skupinou zařízení nejsou jen mobilní telefony, ale i ostatní zařízení připojená k Internetu. Podobné experimenty ukazují, že i zdánlivě neškodné senzory a přístupové technologie mohou být využity ku prospěchu útočníka. Proto je nutné dbát na ochranu osobních informací, byť by se jednalo jen o pohyby či gesta uživatele.

2.5 Bezpečnost platformy Android

Kapitola se zaměřuje na platformě závislé zranitelnosti. Obsahuje informace týkající se virtuální privátní sítě a jejich specifik na Androidu. Dále se zabývá vývojem bezpečnostních opatření systému Android, kde se blíže zaměřuje na Android 7, jenž byl z hlediska změn v oblasti bezpečnosti klíčový. Popsány budou hlavní změny, které dále ovlivnily přístup k jednotlivým procesům. Závěr kapitoly je věnován řešení přístupových práv mobilních

[34] Informace ohledně FaceID: https://www.apple.com/lae/iphone-xs/face-id/
[35] Stránky firmy: https://www.apple.com/lae/

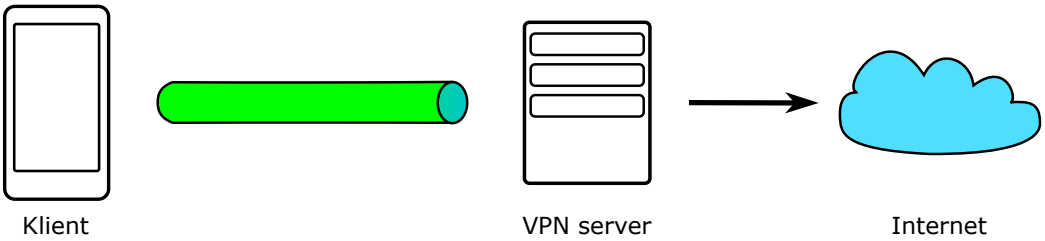


Obrázek 2.4: Schéma akcelerometru a gyroskopu používaného na Android zařízeních. Levý obrázek naznačuje funkcionalitu akcelerometru, který zaznamenává rychlost změny polohy zařízení. Pravá část nastiňuje práci gyroskopu, který zaznamenává rotační změny zařízení dle daných os.

aplikací, včetně vymezení práv uživatele root. Informace ohledně popisované problematiky byly čerpány převážně z knihy *Android Security* [13].

2.5.1 Tunelování internetového připojení pomocí VPN

Virtuální privátní síť (VPN) umožňuje uživatelům zachovat dostatečnou míru soukromí při posílání dat skrze veřejnou síť, např. Internet. Tuneluje komunikaci od klienta k VPN serveru, odkud jsou dále jednotlivé pakety posílány do Internetu. Obrázek 2.5 naznačuje, kdy je možné rozpoznat jeho pravou identitu. Zelená barva značí část, kdy je identita VPN klienta známa, ale data zde prochází šifrovaně. Díky tomu je cíl připojení neznámý. Oproti tomu modrá část znázorňuje prostor, kdy je již identita klienta neznámá, avšak provoz není zabezpečen a je známý cíl požadavků.



Obrázek 2.5: VPN slouží pro zabezpečení komunikace klienta, který chce komunikovat z nedůvěryhodné sítě do Internetu. Obrázek demonstruje připojení zařízení do Internetu, kde se nachází komunikující protistrana. Zelenou barvou je vyznačena část komunikace, kde se dá zjistit pravá totožnost koncového zařízení, ale data jsou zabezpečena. Modrá barva značí prostředí, kde je pravá identita zařízení maskována pomocí VPN, ale data již nemusí být zabezpečena.

Platforma Android umožňuje implementovat vlastní VPN pomocí aplikačního rozhraní *VpnService*[36], aniž by bylo nutné mít oprávnění uživatele root (viz kapitola 2.5.4). Firma Google poskytuje jako příklad VPN s názvem ToyVPN[37], který ukazuje jak ji naimplementovat. Po připojení k vytvořené VPN lze zaznamenávat veškerý síťový provoz mobilního zařízení. V rámci aplikace je pak možné odchytávat jednotlivé pakety a analyzovat, jaká data jsou odesílána či přijímána. To může být využito zejména pro monitorování provozu konkrétní aplikace. Avšak to s sebou nese i riziko podvrhnutí VPN útočníkem, který následně může všechen provoz přesměrovat a získat tím odeslaná osobní data.

2.5.2 Eliminace bezpečnostních rizik

Významné změny v oblasti zabezpečení nastaly ve verzi Android 7. Markantní změnou je posunutí šifrování z úrovně šifrování celých složek na šifrování jednotlivých souborů. Poskytuje to lepší zajištění soukromí jedince v rámci jednoho zařízení, zejména v případě kdy je mobilní zařízení používáno s různými účty. Novinkou je tzv. *direct boot*. Ten umožňuje přístup k aplikacím, které nepracují s osobními údaji uživatele, i když je mobilní zařízení zamčené. Před spuštěním ostatních aplikací je uživatel vyzván, aby prokázal svou totožnost. Tato funkcionalita je dostupná zejména díky šifrování na úrovni souborů. Obsahuje i vylepšení nástroje SELinux[38] spočívajícím v možnosti uzamčení vyhrazeného prostoru aplikace (viz kapitola 2.5.3), čímž se redukuje počet možných rizik. Vylepšení navazuje na změnu v Android 6, kde došlo ke značnému omezení přístupu k procesům aplikací.

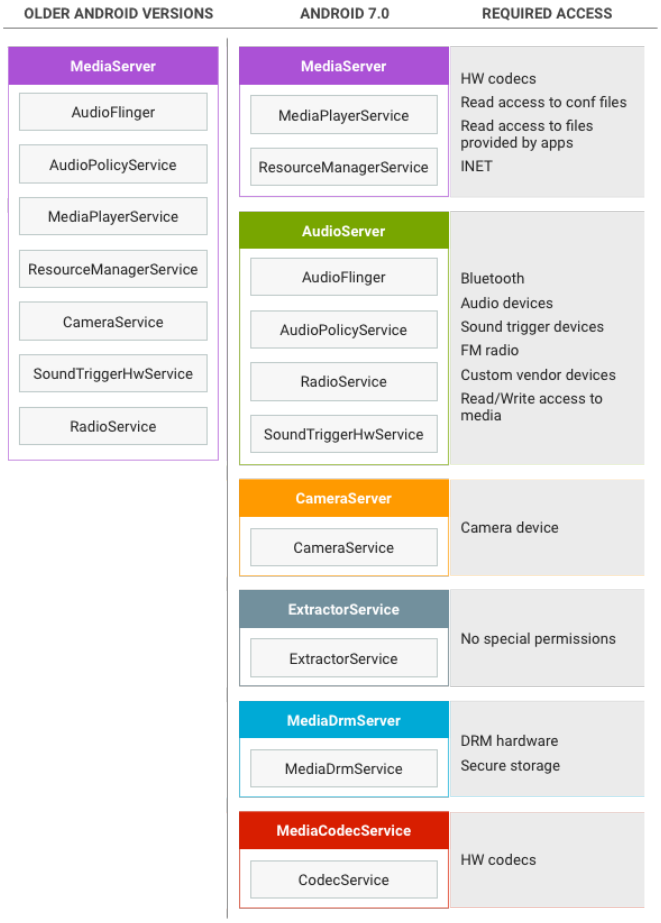
Neméně důležitou změnou je tzv. *Media Framework Hardening*. Pro verze 6 a nižší existoval pouze jeden monolitický proces starající se o všechny multimediální záležitosti zařízení – *MediaServer*. Ten kvůli své komplexnosti vyžadoval nemalé množství oprávnění, byť pracoval jen v rámci svého vyhrazeného prostoru – *sandbox*. Po objevení slabiny, spočívající v možném útoku pomocí přetečení paměti při parsování kodeků a multimediálních formátů, vznikla potřeba tento přístup změnit. Problém byl v tom, že chyba ovlivňovala značné množství zdrojů, i když samotná část pro parsování žádná oprávnění nepotřebuje. Proto byl *MediaServer* od Androidu 7 rozdělen na proces pro parsování, který nemá žádná oprávnění a několik menších procesů s částečnými oprávněními[39]. Toto rozdělení je viditelné na obrázku 2.6, kde jsou barevně odlišeny jednotlivé procesy, které vznikly dekompozicí původního *MediaServer*. Princip rozdělení funkcionality na menší logické bloky s částečnými oprávněními bývá bezpečnější. Nejbezpečnější přístup je, když každý celek zodpovídá jen za jednu funkcionalitu.

[36] *VpnService*: `https://developer.android.com/reference/android/net/VpnService`

[37] Příklad je veřejně dostupný na adrese `https://android.googlesource.com/platform/development/+/master/samples/ToyVpn`

[38] SELinux (Security-Enhanced Linux) je součástí bezpečnostního modelu Androidu a souhrnně označuje všechny poskytované administrativní nástroje systému Linux, které se nacházení v Androidu.

[39] Přesnější informace o procesech: `https://source.android.com/devices/media/framework-hardening`



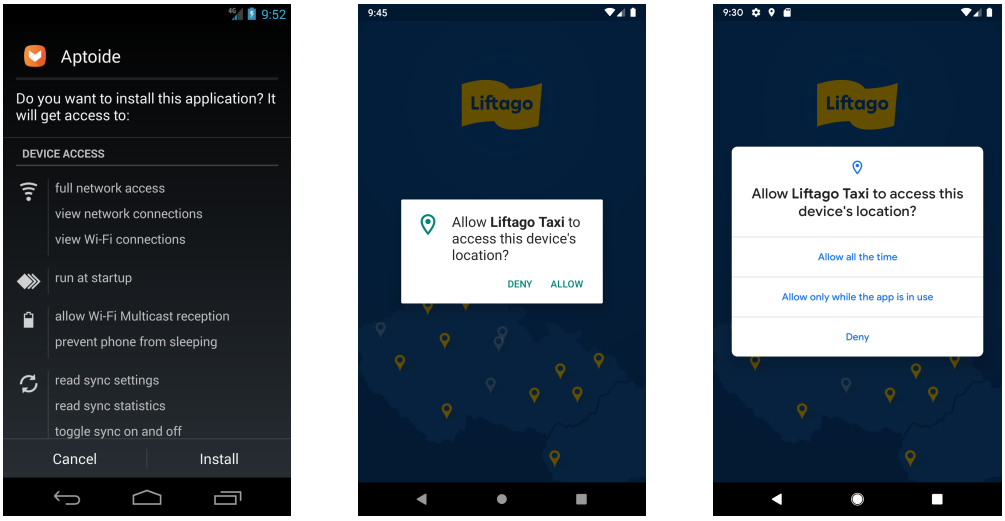
Obrázek 2.6: Obrázek zachycuje architekturní změny procesu *MediaServer*. Ten se od Android 7 rozdělil na více menších procesů, kde každý z nich vyžaduje výrazně méně oprávnění, než potřeboval původní *MediaServer*.

2.5.3 Řešení přístupových práv aplikací

Android je při udělování přístupových práv aplikacím poměrně striktní. Implicitě nemají žádná oprávnění na vykonávání operací, které by mohly ovlivnit ostatní aplikace. Pokud je pro svoji funkčnost potřebují, je nutné, aby si o ně explicitně zažádali. To znamená, že při instalaci či za běhu aplikace bude uživatel vyzván, aby potřebná oprávnění aplikaci udělil. Jednotlivá oprávnění se pak vztahují jen ke konkrétní aplikaci a nikdy nedojde k tomu, že by bylo uděleno oprávnění několika aplikacím současně. Je to díky tomu, že každá aplikace funguje v rámci izolovaného prostoru označovaného jako *sandbox* (volně přeloženo jako „pískoviště“). V něm se pro každou aplikaci spouští jedna instance ART (viz kapitola 3.1.1). Oprávnění, o které aplikace žádá, jsou zaznamenána v Android manifestu. Definují se pomocí `<uses-permission android:name="požadované oprávnění"/>`. Varianty oprávnění, o které lze zažádat, jsou specifikovány v kapitole *Permissions overview* v oficiální dokumentaci[40].

Způsob, jakým je udělováno oprávnění k využívání konkrétních funkcionalit se liší v závislosti na verzi operačního systému. Na obrázku 2.7 jsou snímky aplikací, které vyžadují

[40]Stránka dokumentace: https://developer.android.com/guide/topics/permissions/overview



Obrázek 2.7: Srovnání přístupů k udělování oprávnění na Android zařízeních. Vlevo je nejstarší způsob, který byl využíván až do verze Android 6. Jedná se o udělení všech, pro aplikaci potřebných, oprávnění již při instalaci. Uprostřed je snímek z Android 8. Jedná se o udělení oprávnění za běhu aplikace. Tento přístup je využíván v Android 6 až Android 9. Vpravo je nejnovější způsob, který se objevil spolu s verzí Android Q. V ní se poskytuje oprávnění pouze na určitou akci v závislosti na běhu aplikace. Vlevo je aplikace Aptoide, ostatní snímky jsou z aplikace Liftago.

konkrétní oprávnění. Vlevo se nachází nejstarší typ dialogu, uprostřed je typický dialog pro verze Android 6 až 9 a vpravo je dialog, který bude využíván u nejnovější verze Android Q. Nejvýraznější změna proběhla mezi verzemi Android 5 a 6. Zatímco pro Android 5 a nižší byla oprávnění udělována hromadně hned při instalaci, tak od verze 6 se aplikace nainstaluje s minimálními oprávněními a na ostatní se doptává pomocí dialogu až za běhu. Díky tomuto opatření je jasnější, která operace vyžaduje jaké oprávnění. Uživatel má pak větší přehled o tom, v jaké části aplikace je konkrétní funkcionalita požadována.

Z uživatelského pohledu se oprávnění u dosavadních verzí Android systému jeví jako jednorázová akce. Běžný uživatel ve snaze zpřístupnit požadovanou funkcionalitu tyto oprávnění potvrdí a při následném používání aplikace se o ně nezajímá. Tento koncept by se mohl změnit s nástupem nové verze operačního systému. U Android Q přibude ikonka upozorňující uživatele, že některé spuštěné aplikace využívají potencionálně nebezpečné senzory. Má se to týkat zejména využívání mikrofonu, fotoaparátu a polohy zařízení, přičemž ke každému z nich si aplikace musí vyžádat potřebná oprávnění. Kliknutím na ikonku se uživatel dozví, jaký senzor je aktivně využíván a která aplikace ho vyžaduje. Jinak řečeno, která aplikace aktivně využívá své oprávnění pro ovládání konkrétního senzoru. Další změnou je možnost dočasného povolení přístupu k poloze zařízení. To je možné udělit pouze za běhu aplikace a je aktivní jen tehdy, když se aplikace nachází na popředí. Pokud se však uživatel rozhodne udělit trvalé povolení přístupu k poloze zařízení, pak o tomto faktu bude za běhu aplikace informován pomocí upozornění v záhlaví. Kliknutím na toto upozornění se uživateli otevře nastavení oprávnění, kde lze aplikaci udělené oprávnění snadno zrušit. Cílem těchto upozornění je zejména snaha informovat uživatele, že některá ze spuštěných aplikací využívá jeho informace o poloze, přestože běží pouze na pozadí.

2.5.4 Práva uživatele root

Získáním práv uživatele *root*[41] se otevírají nové možnosti, jak si přizpůsobit chování i vizuální stránku svého Android zařízení. Na Internetu koluje značné množství návodů, jak tyto práva získat. Dají se rozdělit obecně na dva přístupy, tj. s využitím osobního počítače, či pouze za pomoci mobilního zařízení samotného. Mezi benefity patří možnost mazat či upravovat systémové aplikace, libovolně manipulovat se soubory nebo třeba přetaktovat zařízení, kvůli získání vyššího výkonu. Dá se říct, že možnosti root uživatele jsou při správě systému neomezené. Velmi zajímavou možností je vymanit aplikaci z jejího izolovaného prostoru (viz *sandbox* v kapitole 2.5.3) a získat tím možnost přistupovat k ostatním aplikacím či k chráněným systémovým funkcím. Aplikace, které tohoto přístupu využívají, slouží například pro správu souborů, správu procesů běžících na pozadí či dokonce pro čtení databází ostatních aplikací.

V případě zájmu o nové možnosti a aplikace, které tento přístup přináší, je nutné zvážit také hrozby plynoucí z velkého množství práv. Napadení takového mobilního zařízení či nainstalování škodlivé aplikace může vést ke ztrátě veškerých uložených osobních dat nebo dokonce ke ztrátě kontroly nad mobilním zařízením. Avšak nemusí jít přímo o napadení systému zvenčí. Velkým nebezpečím je i nedostatečně znalý uživatel, který omylem provede změnu například v souborovém systému. Zvýšení práv uživatele proto zůstává spíše doménou nadšenců pro danou platformu. Pro běžného uživatele nejsou benefity root uživatele dostatečně velké oproti hrozícímu nebezpečí.

[41] Uživatel root je obecně v Linux systémech uživatel s UID=0, který disponuje neomezenými oprávněními.

Kapitola 3

Použité technologie

Kapitola je zaměřena zejména na popis platformy Android, včetně nadstavby Android Jetpack. Dále se dotkne i implementačního jazyka Kotlin, který se stává stále oblíbenějším, zejména při vývoji Android aplikací. V závěru kapitoly je popsána struktura pcap souboru, který slouží pro ukládání síťových toků.

3.1 Operační systém Android

Operační systém Android spatřil světlo světa 5. listopadu 2007, kdy byla vydána první beta verze. Vyvíjí ho společnost Google a koncem roku 2019 by měla vyjít stabilní verze nejnovějšího Android Q, který bude patnáctým operačním systémem pojmenovaným podle sladkostí, nicméně jeho název ještě nebyl oficiálně oznámen.

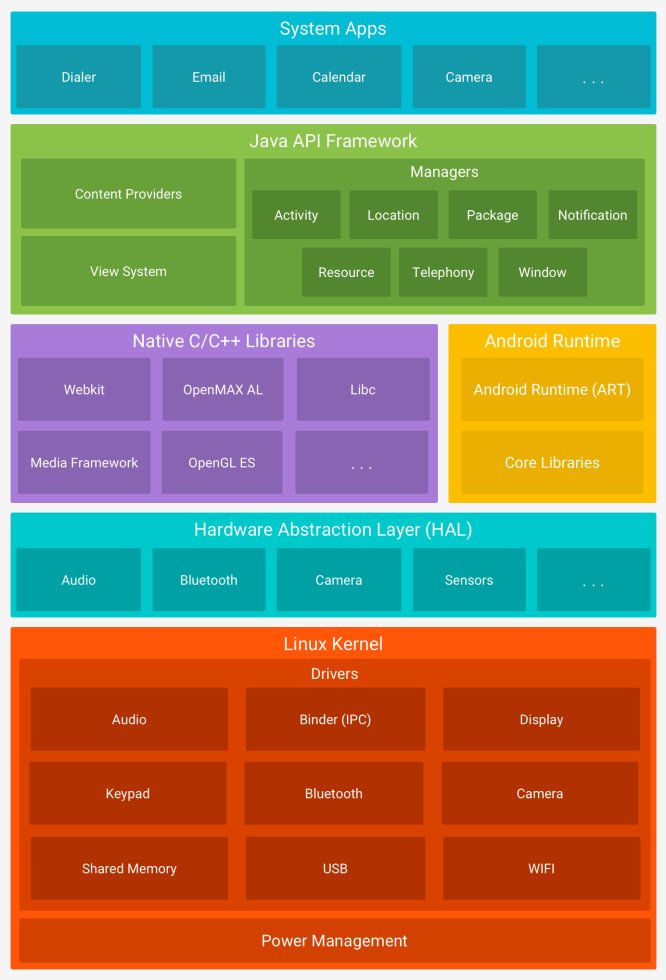
V této kapitole budou popsány hlavní části architektury operačního systému Android. Zaměří se na možnosti poskytované Linux kernelem, ze kterého vychází jádro Androidu. Neméně důležitou částí budou informace týkající se architektury aplikací pro Android a pojmy jako je aktivita nebo fragment.

3.1.1 Architektura operačního systému

Architektura operačního systému Android se dá rozdělit na pět základních úrovní. Obsah jednotlivých úrovní i jejich hierarchie je přehledně vidět na obrázku 3.1, který je převzatý z oficiální dokumentace operačního systému Android[1]. Na nejnižší z nich se nachází jádro systému, které vychází z Linux kernelu. Avšak Android obecně se nedá považovat za jeden z linuxových systémů. Důvodem je zejména to, že neimplementuje veškeré náležitosti, které jsou pro ně typické. Nesetkáme se například s implementací *GNU C Library*[2] nebo s *X Window System*[3]. Přesto však existuje závislost verzí Android jádra a Linux kernelu, která je zobrazena v tabulce B.1 v příloze B. Druhou úrovní je mezivrstva mezi hardware a software Android zařízení, označovaná jako *Hardware Abstraction Layer*. Obsahuje moduly poskytující rozhraní pro komponenty, jako je fotoaparát, GPS, bluetooth, atp.

Následují knihovny a běhové prostředí Androidu. Je zde zabudovaná podpora C a C++ knihoven, které se dají používat skrze aplikační framework, přičemž většina z nich funguje bez dodatečných modifikací. Běhové prostředí se skládá ze dvou částí. Jednou je výkonný prvek (ART nebo Dalvik VM) a druhým jsou knihovny jádra:

[1] Viz https://developer.android.com/guide/platform
[2] Kompletní popis knihovny: https://www.gnu.org/software/libc/manual/html_mono/libc.html
[3] Bližší informace ohledně *X Window System*: https://en.wikipedia.org/wiki/X_Window_System



Obrázek 3.1: Architektura Android systému rozdělená do jednotlivých úrovní. Celkově se dá počítat, že je systém rozdělen na pět úrovní, neboť nativní knihovny a běhové prostředí spolu velmi úzce souvisí.

- *Dalvik Virtual Machine* zkráceně *Dalvik VM* je virtuální stroj optimalizovaný pro zařízení s malým výkonem. Zodpovídá za vykonávání programu ve formátu DEX. Ten je získán kompilací z Java třídních souborů (*Java class files*), do kterých je uložen kompilovaný zdrojový kód psaný v Javě.
- Alternativou k *Dalvik VM* je ART, čili *Android Runtime*. Narozdíl od *Dalvik VM* vytváří z DEX nativní kód, který následně vykonává.
- Knihovny jádra (*Core Libraries*) zprostředkovávají většinu funkcionality programovacího jazyka Java.

Čtvrtou úrovní je aplikační framework, který zapouzdřuje funkcionality operačního systému Android. Skrze něj lze pohodlně pracovat s jednotlivými komponentami systému jako s ucelenými bloky[4]. Poslední vrstva obsahuje řadu aplikací, které jsou v této platformě zabudovány. Jedná se o kalendáře, internetové prohlížeče nebo například aplikaci pro posílání SMS. Tyto aplikace však nemají výhradní postavení oproti ostatním, uživatelem

[4]Blíže jsou popsány v sekci *Java API Framework*: https://developer.android.com/guide/platform/

nainstalovaným, aplikacím. Ve výsledku to znamená, že kteroukoliv z těchto aplikací lze vyměnit za jinou dostupnou aplikaci, která od té doby bude považována za výchozí.

3.1.2 Jádro systému

Jak již bylo zmíněno, základem Android jádra je Linux kernel. Tento fakt poskytuje zajímavé možnosti využití Androidu. Jednou z nich je například podpora běhu linuxových aplikací, proveditelná i bez práv uživatele *root* (blíže v kapitole 2.5.4). Jedna z aplikací, která tuto možnost poskytuje, se jmenuje UserLAnd[5]. Další z možností je vykonávání unixových příkazů. Seznam předinstalovaných příkazů se liší v závislosti na konkrétním zařízení i verzi operačního systému. Obecně však platí, že při využití práv uživatele root lze požadované příkazy doinstalovat. Seznam všech dostupných příkazů lze získat v terminálu zařízení[6] pomocí příkazu `ls /system/bin` případně jen `ls`.

3.1.3 Architektura Android aplikací

Android aplikace se z pohledu struktury projektu skládá z několika nezbytných součástí. Je to *AndroidManifest*, *gradle* skript, základní aktivita a zdroje aplikace. K tomu se mohou přidávat další aktivity, fragmenty, případně pak testy a další třídy. *AndroidManifest* je jádrem aplikace. Je v něm definována základní aktivita, jméno balíčku aplikace, její název i ikona. Dále jsou zde definována potřebná oprávnění (bližší informace k oprávněním jsou v kapitole 2.5.3) a explicitně vyjmenované komponenty aplikace, tj.:

- **Aktivita** – deklaraci obaluje element `<activity>`, poskytuje funkcionalitu pro jednu nezávislou obrazovku v aplikaci,
- **Service** – deklaraci obaluje atribut `<service>`, využívá se pro výpočetně náročné či dlouho běžící operace,
- **BroadcastReceiver** – deklaraci obaluje atribut `<receiver>`, dokáže přijímat požadavky všesměrového vysílání a umožňuje aplikaci na ně reagovat,
- **ContentProvider** – deklaraci obaluje atribut `<provider>`, je určen pro sdílení v něm uložených dat mezi různými aplikacemi.

Zmíněné elementy patří mezi nejdůležitější, nicméně povinné jsou jedině atributy `<manifest>` a `<application>`. V souboru *AndroidManifest* je možné využít podstatně více elementů. Jejich celkový přehled je dostupný na stránce oficiální dokumentace k *AndroidManifest*[7]. *Gradle* skriptů je v projektu aplikace běžně hned několik. Existuje zde vždy jeden hlavní skript pro celý projekt, jeden skript pro mudul aplikace a případně další skripty pro ostatní implementované moduly. Hlavním účelem všech *gradle* skriptů je definovat závislosti na externích knihovnách. Skript, který náleží samotné aplikaci, určuje i výchozí konfiguraci, která obsahuje id aplikace, minimální verzi SDK, cílovou verzi SDK, kód verze aplikace, název verze aplikace a verzi SDK využívanou pro kompilaci. Mimo tyto údaje je zde důležité definovat využívané *gradle* pluginy, pokud jsou potřeba.

[5] Aplikace *UserLAnd* je dostupná na GooglePlay s odkazem https://play.google.com/store/apps/details?id=tech.ula.

[6] Na trhu jsou k dispozici aplikace, které umožňují přístup k terminálu. Dobře hodnocenou aplikací je například *Terminal Emulator*: https://play.google.com/store/apps/details?id=jackpal.androidterm.

[7] Viz https://developer.android.com/guide/topics/manifest/manifest-intro

Aktivita

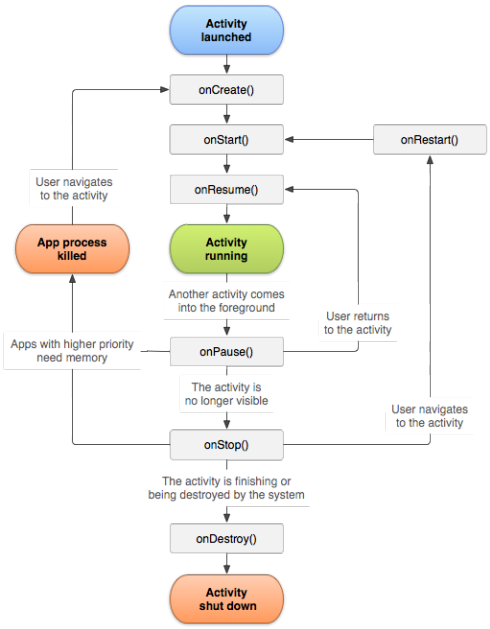
Aktivita reprezentuje základní komponentu pro vytvoření Android aplikace. Z uživatelského pohledu je aktivita jedna obrazovka. Stejně, jako se v některých ostatních programovacích jazycích začíná kód vykonávat od funkce *main()*, tak v Android aplikacích se začíná od základní aktivity. Ta musí být definována, jako počáteční aktivita v *AndroidManifest*. Dobrým zvykem je, že se v této aktivitě provádí jen minimum věcí. Většinou to jsou nezbytné inicializace a vyvolání další aktivity (či fragmentu), jejíž UI již bude zobrazeno uživateli.

Každá aktivita má životní cyklus, který programátorovi pomáhá při práci s událostmi, jenž v aplikaci nastaly. Pro představu, jak spolu jednotlivé fáze tohoto cyklu souvisí slouží obrázek 3.2a. Jedná se o metody, které jsou potřebné pro zobrazení správného obsahu a zajištění korektního chodu aplikace. Volají se interně, v závislosti na změně stavu aplikace. Každá z nich má své specifické využití a programátor by je měl přepisovat, jen pokud je to nezbytné pro zajištění funkcionality. Většinou se v praxi přepisují metody *onCreate()*, *onResume()*, *onPause()* a *onStop()*, kvůli řešení stavu aplikace navázaného na její skrytí či obnovení. Aktivita má 7 základních metod životního cyklu:

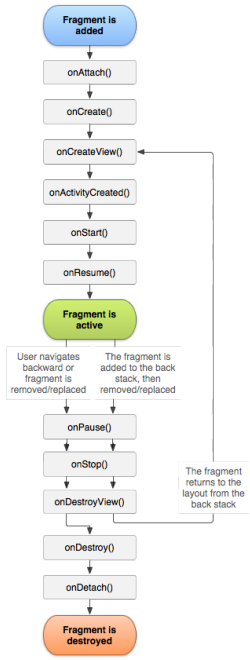
- *onCreate()* – při prvním vytvoření aktivity, je vhodným místem pro nastavení statického obsahu,
- *onStart()* – při zviditelnění aktivity, tj. při obnovení z pozadí či po vytvoření,
- *onResume()* – při obnovení, dříve než aktivita začne komunikovat s uživatelem,
- *onPause()* – při pozastavení aktivity a jejím umístění na pozadí nebo když není celá viditelná pro uživatele, např. když je aspoň z části překryta dialogem,
- *onStop()* – při přechodu z popředí, tj. dříve než ji celou překryje jiná obrazovka či je zničena,
- *onRestart()* – při obnovení aktivity a jejím umístění do popředí,
- *onDestroy()* – při zničení aktivity.

Fragment

Fragment není nezbytnou součástí Android aplikace. Umožňuje rozdělení obrazovky na více logických celků, které jsou vždy součástí aktivity, tzn. fragment nemůže být bez aktivity zobrazen. Platí mezi nimi vztah, že jedna aktivita může mít $0 \ldots N$ fragmentů, přičemž jednotlivé fragmenty mohou být znovu použity ve více aktivitách. Společně kooperují především v konceptu *master/detail*, který byl vytvořen zejména pro tablety a ostatní zařízení se širokým displejem. Využívá zpravidla jedné aktivity a dvou či více fragmentů. Ty jsou připojeny k aktivitě a vzájemně kooperují ve vztahu *master/detail*. V případě tabletů a obdobných zařízení se pak fragmenty, zobrazují vedle sebe v rámci dané aktivity. U menších zařízení se při interakci s *master* fragmentem zobrazí *detail* fragment na nové obrazovce. Využití *master* a *detail* fragmentu na jedné obrazovce se označuje jako *two pane* rozložení (viz obrázek 3.3b), tj. rozložení do dvou sloupců. Pokud obrazovka obsahuje jen jednu komponentu, tak se označuje jako *one pane* (viz obrázek 3.3a).



(a) Schéma životního cyklu aktivity v operačním systému Android.



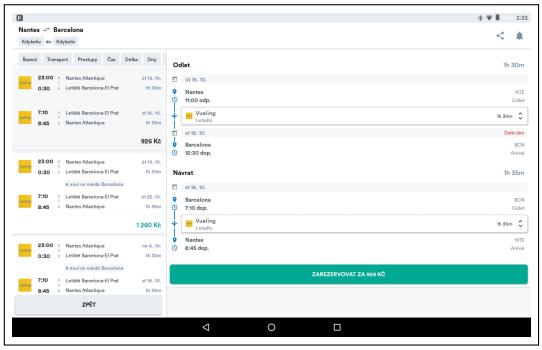
(b) Schéma životního cyklu fragmentu v operačním systému Android.

Obrázek 3.2: Životní cykly aktivity i fragmentu se v mnoha bodech shodují. Cyklus fragmentu obsahuje více metod díky jeho závislosti na existenci aktivity.

Podobně jako aktivita i fragment má svůj životní cyklus, nicméně je o něco komplikovanější. Je to způsobeno tím, že je fragment existenčně závislý aktivitě, ke které je připojen. Kvůli tomu musí korelovat svůj životní cyklus vzhledem ke stavu aktivity. Jednotlivé metody životního cyklu fragmentu a jejich následnost je schématicky popsána na obrázku 3.2b. Stejně jako u aktivity i u fragmentu platí, že původní implementace metod životního cyklu by se měly přepisovat pouze, pokud je to pro běh aplikace nezbytné. Funkcionalita jednotlivých metod je podobná, jako u aktivity, avšak fragment implementuje některé metody navíc. Životní cyklus fragmentu má o 4 metody víc, než cyklus aktivity. Celkově jich má 11:

- *onAttach()* – při připojení fragmentu k aktivitě, ještě před vykreslením fragmentu,
- *onCreate()* – při prvním vytvoření fragmentu,
- *onCreateView()* – při prvním vykreslení fragmentu,
- *onActivityCreated()* – při dokončení metody *onCreate()* aktivity, ke které je fragment připojen, je vhodným místem pro nastavení statického obsahu,
- *onStart()* – při zviditelnění fragmentu,
- *onResume()* – při obnovení fragmentu,
- *onPause()* – při pozastavení fragmentu a jeho umístění na pozadí či při překrytí jeho části jiným prvkem, např. dialogem,
- *onStop()* – při zneviditelnění fragmentu,

(a) *One pane* rozložení obrazovky. Pro tablet je nevhodné, protože nevyužívá celý prostor displeje ke zobrazení podstatných informací. Nicméně je využívané u mobilních telefonů a zařízení s menší velikostí displeje.



(b) *Two pane* rozložení obrazovky. Pro tablet je vhodné díky tomu, že zvládne vyplnit celý prostor displeje. Je vhodné, když je potřeba zobrazit uživateli aktivitu i její detail na jedné obrazovce. Vlevo lze vidět panel aktivity, vpravo je detail vybraného spojení.

Obrázek 3.3: Rozdíl mezi rozloženími obrazovky v Android aplikaci. Vlevo je pro tablet méně vhodné rozložení. Vpravo je vhodné využití konceptu *master/detail*. Snímky jsou pořízeny z aplikace Kiwi.com.

- *onDestroyView()* – před finálním zničením fragmentu, je vhodným místem pro zavolání kódu, který provádí úpravy stavu programu, před opuštěním obrazovky,
- *onDestroy()* – před zničením fragmentu, vhodné pro vyčištění stavu fragmentu,
- *onDetach()* – při zničení fragmentu, když už není připojen ke konkrétní aktivitě.

3.1.4 Android Jetpack

Andorid Jetpack[8] je kolekce čtyř skupin softwarových komponent, které zjednodušují vývoj Android aplikací. Jsou pojmenované jako: *foundation*, *architecture*, *behaviour* a *UI*. Komponenty jsou kolekcí knihoven, které lze v projektu využít samostatně. Jsou navrženy tak, aby byly schopné kooperace. Celá kolekce Andorid Jetpack využívá jako implementační jazyk Kotlin a z hlediska podpůrné knihovny Androidu (*Android Support Library*) využívá *AndroidX*, nicméně zachovává zpětnou kompatibilitu s *v7 appcompat library*.

Pomocí nových knihoven, které Andorid Jetpack obsahuje lze řešit i základní problémy, jako je například navigace v rámci aplikace. Z tohoto pohledu je zajímavým řešením využití třídy `NavHostFragment`, která poskytuje zjednodušení navigace v rámci celé aplikace. Pomocí ní lze implementovat *bottom navigation*, což je rozložení aplikace, které má ovládací prvky umístěny v liště v dolní části obrazovky. Dále řeší animace mezi jednotlivými obrazovkami a automatizuje přepínání fragmentů. Využívá konceptu *single activity*, což

[8]Podrobnosti na stránce: https://developer.android.com/jetpack/

znamená, že je celá aplikace je řízena jednou aktivitou. Zbylé obrazovky jsou vytvářeny pomocí fragmentů.

3.2 Kotlin

Kotlin[9] je silně typovaný programovací jazyk vyvinutý společností JetBrains[10]. První stabilní verze byla publikována 15. února 2016[11], avšak s jeho vývojem se začalo již v roce 2011. Dnes používané verze vznikly díky práci vývojářů v JetBrains a komunitě nadšenců, kteří tento jazyk testovali přímo na svých projektech. Již od počátku byl zamýšlen, jako nástupce jazyka Java[12]. Díky tomu je přechod mezi nimi snadný a dokáží spolu kooperovat v rámci jednoho projektu. Vzájemnou kombinaci při vývoji umožňuje i IDE od JetBrains. Jejich Android Studio[13] poskytuje přímo možnost překompilování Javy do Kotlinu, či naopak. Podstatné informace, včetně ukázek kódu lze najít v knize *Kotlin in Action* [9], která byla významným zdrojem informací uvedených v této kapitole.

Podle analýzy dat nasbíraných firmou Pusher[14] roste obliba Kotlinu již od vydání první stabilní verze. Rozšířil se zejména mezi vývojáři Android aplikací, což je zapříčiněno zejména tím, že reflektuje potřeby vývoje pro mobilní platformy lépe než Java. Z analýzy vyplývá, že z programátorského hlediska je nejdůležitějším faktorem vedoucím k využívání tohoto jazyka jeho řešení proměnných s `null` hodnotou. Zatímco Java umožňovala do kterékoliv proměnné konkrétního datového typu uložit i `null` hodnotu, Kotlin tento přístup neumožňuje. Při deklaraci proměnné v Kotlinu je nutné určit, zda do ní bude možné uložit hodnotu `null`, nebo ne. Toto opatření eliminuje zejména chyby aplikací, kdy spadly s výjimkou upozorňující na snahu přistoupit na nulový prvek pomocí ukazatele, tj. `NullPointerException`.

```
class Application {
 val id = 0
 val name: String?
 var icon: Drawable? = null
}
```

Výpis 3.1: Ukázka třídy `Application` v jazyce Kotlin. Proměnná `id` nesmí obsahovat hodnotu `null`. Oproti tomu `name` a `icon` tuto hodnotu obsahovat mohou, což je vyjádřeno pomocí symbolu otazníku. Na základě přiřazení celočíselné hodnoty je při překladu vyhodnocen datový typ proměnné `id` jako `Int`.

Výpis 3.1 slouží, jako ukázka zmiňované funkcionality Kotlinu, včetně základního nástinu syntaxe. Třída `Application`, uvozená klíčovým slovem `class`, obsahuje tři proměnné – `id`, `name` a `icon`. Ty se obecně dělí na dva typy – `var` a `val`. `Var` je proměnná, které se může měnit hodnota. Oproti tomu proměnné uvozené slovem `val` lze přiřadit hodnotu pouze jednou. Obdobnou významovou hodnotu má v Javě klíčové slovo `final`, které se při vytváření proměnné umisťuje před její datový typ.

[9] Viz oficiální stránky: https://kotlinlang.org/
[10] Stránky společnosti: https://www.jetbrains.com/
[11] Dle článku z oficiálních stránek JetBrains: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/
[12] Viz oficiální stránky tohoto jazyka: https://www.java.com/en/
[13] Nástroj vytvořený primárně pro Android projekty. Viz https://developer.android.com/studio/
[14] Výsledek analýzy v grafech lze najít na stránkách firmy: https://pusher.com/state-of-kotlin

Informace, zda se do dané proměnné dá uložit nulová hodnota nebo ne, je určena pomocí symbolu otazníku. Do proměnných `name` a `icon` lze uložit `null`, do `id` nikoliv. Výsledný datový typ proměnné `id` bude `Int`, což se rozhodne podle přiřazené celočíselné hodnoty 0. Ukázka stejného příkladu v Javě je ve výpisu 3.2. Lze si povšimnout již zmíněných rozdílů v podobě syntaxe a možností jazyka. Navíc tento výpis podtrhuje fakt, že Kotlin umí vyhodnotit datový typ při překladu a není nutné jej explicitně psát.

```
class Application {
 int id = 0;
 String name;
 Drawable icon = null;
}
```

Výpis 3.2: Ukázka třídy *Application* v jazyce Java. Všechny proměnné zde monou obsahovat hodnotu `null`, neboť Java striktně nezakazuje její uložení do kterékoliv proměnné.

Od Javy se v Kotlinu liší nejen syntaxe proměnných, ale i metod. Výpis 3.3 obsahuje ukázku *getteru* a *setteru* proměnné *name*, jenž byla již zmíněna ve výpisu 3.1. Definice těchto dvou metod je navázána na proměnnou, jenž se nad nimi nachází. Výhodou tohoto přístupu je, že se zpřehlední nastavování a získávání hodnoty z dané proměnné. V Javě bylo nutné explicitně vyvolat metody *get()* či *set(value)*. V Kotlinu se tyto metody volají interně. To znamená, že se programátor o jejich volání nemusí starat. V případě přiřazení do proměnné se provolá metoda *set(value)*, tj. v případě provolání kódu `application.name = "AppCheck"`. Naopak, pokud chceme hodnotu získat, tak se při zavolání `val oldName = application.name` interně provede metoda *get()*. Pro Kotlin typickým klíčovým slovem, používaným v *getterech* či *setterech*, je *field*. To odkazuje přímo na paměťové místo, kde je daná proměnná uložená a existuje jen v rámci této metody. Pro získání či nastavení hodnoty proměnné je v Javě nutné vytvořit dvě metody, které se od ostatních metod v zásadě neliší. Jejich identifikačním znakem se stala předpona v názvu metody. Jedná-li se o *setter*, tak se metoda označuje jako *setNazevPromenne()*, kde sémantickou předponou je slovo *set*. Obdobně u *getteru* se využívá slovo *get*, například *getNazevPromenne()*. Tyto předpony nemají z hlediska překladu programu význam. Slouží zejména pro orientaci v kódu. Ve výpisu 3.4 jsou v Javě vytvořeny srovnatelné metody jako ve výpisu 3.3. Z ukázky je vidět, že Kotlin díky svému přístupu poskytuje kratší možný zápis. Avšak jeho největší výhodou je již zmíněné interní provolávání metod. Pro nastavení hodnot v Javě je nutné explicitně zavolat metodu *setNazevPromenne()*, pro získání hodnoty *getNazevPromenne()*. Provolání těchto metod pak vypadá jako `application.setName("AppCheck");` či `String oldName = application.getName();`.

```
var name: String?
 get() = randomName()
 set(value) { field = value }
```

Výpis 3.3: Ukázka *getteru* a *setteru* definovaných pro proměnnou `name` v jazyce Kotlin. Hlavní výhodou tohoto přístupu je minimalizace kódu a přehlednost.

```
String name;
public String getName() { return randomName(); }
public void setName(String value) { name = value; }
```

Výpis 3.4: Ukázka *getteru* a *setteru* definovaných pro proměnnou `name` v jazyce Java. Hlavní výhodou tohoto přístupu je minimalizace kódu a přehlednost.

Kotlin poskytuje více funkcionalit týkajících se řešení `null` hodnot. Ty umožňují programátorovi s nimi pracovat do jisté míry bezpečně. Jednou z nich je volat metody na proměnných s možností uložení hodnoty `null`, aniž by došlo k vyvolání `NullPointerException` výjimky, jako by tomu bylo v Javě. Volání metody `application.getName()`, kde je proměnná `application` rovna `null`, by se v Kotlinu přepsalo pomocí bezpečného operátoru „?.“, jako `application?.getName()`. Zmíněný výraz by pak vrátil hodnotu `null`. Ostatní funkcionality, jako – „elvis“ operátor ?:, `data class`, funkce vyšších řádů, Kotlin coroutines, atp. – jsou uvedeny v knize *Kotlin in Action* [9], případně na oficiálních stránkách[15] nebo v řadě tutoriálů dohledatelných na Internetu.

3.3 Formát souboru pcap

Dle textu[16] na stránkách organizace IANA[17] byl pcap vytvořen autory nástroje tcpdump[18], jako vhodný formát souboru pro zaznamenávání síťového toku. Pánové V. Jacobson, C. Leres a S. McCanne ho nejdříve zkoušeli na BSD UNIX, později se rozšířil i na ostatní operační systémy. V současnosti je spojovaný především s nástrojem Wireshark[19], který slouží mimo jiné pro filtrování a zaznamenávání síťového toku.

Soubor uložený v pcap formátu má pevně danou strukturu[20]. I když se pcap stále vyvíjí, tak struktura souboru zůstává nezměněna. Prvním záznamem v souboru je vždy globální hlavička. Ta obsahuje základní informace o zaznamenaném toku. Struktura této hlavičky je ukázána ve výpisu 3.5. Skládá se ze 7 položek, kde:

- `magic_number` – určuje formát souboru a pořadí bytů,
- `version_major` – určuje hlavní verzi formátu,
- `version_minor` – určuje vedlejší verzi formátu,
- `thiszone` – určuje korekci času mezi GMT (UTC) a lokální časovou zónou,
- `sigfigs` – určuje přesnost časových razítek v záznamu,
- `snaplen` – určuje délku záznamu,
- `network` – určuje typ hlavičky linkové vrstvy.

[15] Viz oficiální stránky: https://kotlinlang.org/
[16] Text z roku 2011: https://www.iana.org/assignments/media-types/application/vnd.tcpdump.pcap
[17] Viz https://www.iana.org
[18] Viz oficiální stránky: http://www.tcpdump.org
[19] Viz https://www.wireshark.org
[20] Informace o pcap byly čerpány z https://wiki.wireshark.org/Development/LibpcapFileFormat

```
typedef struct pcap_hdr_s {
    guint32 magic_number;  /* magic number */
    guint16 version_major; /* major version number */
    guint16 version_minor; /* minor version number */
    gint32 thiszone;    /* GMT to local correction */
    guint32 sigfigs;    /* accuracy of timestamps */
    guint32 snaplen;    /* max length of captured packets, in octets */
    guint32 network;    /* data link type */
} pcap_hdr_t;
```

Výpis 3.5: Základní struktura globální hlavičky pcap souboru.

Na globální hlavičku je napojeno $0 \ldots N$ bloků týkajících se zaznamenaných paketů. Každý blok se skládá z hlavičky a dat paketu. Hlavičky obsahují informace určující, o který paket se jedná. Struktura informací je patrná z výpisu 3.6. Význam jednotlivých položek je:

- `ts_sec` – určuje časové razítko, kdy byl paket zachycen,
- `ts_usec` – určuje časové razítko v milisekundy, kdy byl paket zachycen, slouží jako upřesnění času uloženého v `ts_sec`,
- `incl_len` – určuje počet bytů, které byly zachyceny a uloženy do souboru, avšak nikdy by to nemělo být číslovětší než snaplen v globální hlavičce,
- `orig_len` – určuje původní délku paketu, není limitované velikostí snaplen.

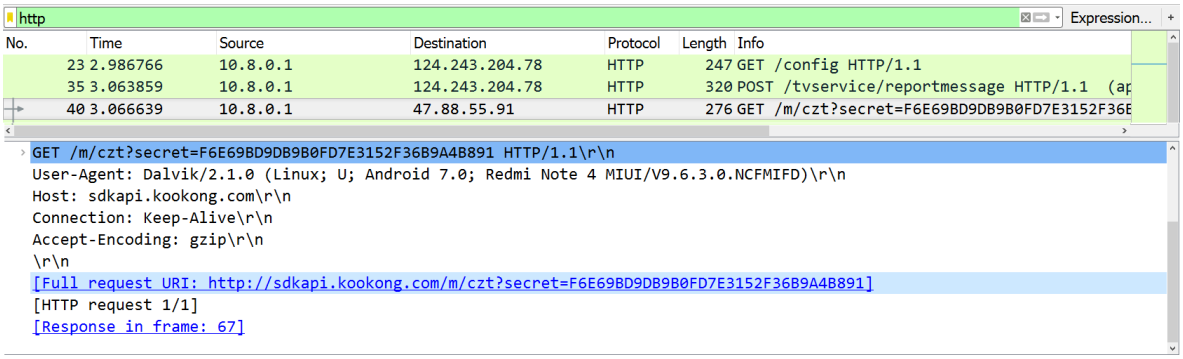
```
 typedef struct pcaprec_hdr_s {
    guint32 ts_sec;     /* timestamp seconds */
    guint32 ts_usec;    /* timestamp microseconds */
    guint32 incl_len;    /* number of octets of packet saved in file */
    guint32 orig_len;    /* actual length of packet */
} pcaprec_hdr_t;
```

Výpis 3.6: Základní struktura hlavičky paketu v pcap souboru.

Kapitola 4

Návrh aplikace

Nápad vedoucí ke vzniku aplikace jsem dostala při monitorování síťové komunikace Android zařízení. Během něj jsem se pomocí nástroje Wireshark snažila zjistit, zda-li jedna konkrétní aplikace odesílá data. Jednalo se o firemní aplikaci u které jsem znala IP adresu serveru, na který měla komunikovat. Při monitorování mě zaujal hned třetí zachycený paket, který obsahoval tajemství aplikace zaslané v čitelné podobě pomocí protokolu HTTP. Zachycené tajemství včetně dalších informací v hlavičce je na obrázku 4.1. Tento nález mě přivedl na myšlenku, že bych ráda zjistila, která aplikace tak neopatrně zachází se svým tajemstvím. Přimělo mě to se zamyslet nad tím, kolik jiných aplikací se může chovat tímto způsobem, případně kolik jich obdobně odesílá citlivá data uživatelů. S realizací aplikace, která bude schopná tyto problémy evidovat, mi pomohl prof. Ing. Adam Herout, Ph.D., který mi umožnil toto téma zpracovat jako diplomovou práci.



Obrázek 4.1: Záznam zachycený během 5 minutového monitorování síťového toku mobilního zařízení. Ve třetím řádku záznamu lze vidět nezabezpečeně odeslané tajemství neznámé mobilní aplikace, která komunikovala na adresu *sdkapi.kookong.com*.

Primárním cílem navrhované aplikace je poskytnout uživateli informaci ohledně bezpečnosti ostatních aplikací bez nutnosti speciální úpravy systému Android. Tj. bez nutnosti zvýšení práv běžného uživatele na úroveň *root*, což je blíže popsáno v kapitole 2.5.4. Přehledně informovat o tom, kam a ze které aplikace jsou pakety odesílány, jaké protokoly využívají a případně další informace, jenž by mohly být podstatné pro určení bezpečnosti aplikace. Většina požadovaných funkcionalit není dostatečně dokumentovaná či probádaná. Díky tomu má aplikace experimentální charakter a její návrh vznikal iterativně.

4.1 Srovnání již existujících aplikací

Spolu s vytyčením cíle navrhované aplikace bylo nutné zjistit, zda-li jsou požadované funkcionality proveditelné. Podařilo se mi najít několik aplikací, které se aspoň částečně podobají navrhované aplikaci. Nicméně takovou, která by uživateli poskytovala přehled o míře zabezpečení ostatních aplikací, jsem nenašla. Konkrétně se zaměřím na srovnání navrhované aplikace a tří již existujících, které byly z testovaných aplikací nejvíce použitelné pro srovnatelný účel. Vybranými aplikacemi jsou: Netstat Plus[1], Packet Capture[2] a NetCapture[3]. Pro lepší představu o tom, co konkrétní aplikace umí, jsem sestavila tabulku funkcionalit 4.1 a jednotlivé položky pro porovnání jsem vybrala tak, aby reflektovaly nezbytné součásti aplikace.

UDP a TCP indikují, které transportní protokoly jsou sledovány. Stejně jako IPv4 a IPv6 říká, zda jsou tyto protokoly podporovány. Cílový a zdrojový port indikuje, zda lze pomocí aplikace zjistit jejich čísla. Čitelný obsah HTTPS paketů určuje, zda je možné získat dešifrovaný obsah paketů zaslaných pomocí HTTPS. Informace o aplikaci říkají, zda je zjistitelný název aplikace, její verze a název balíčku. To je užitečné při přesné identifikaci aplikace, kterou chceme sledovat. Zobrazení připojení poskytuje informace o aktuálním síťovém připojení, včetně jeho aktuálního stavu. Zachycení poslaných paketů určuje, zda je aplikace schopna zaznamenat probíhající komunikaci. To je úzce propojené s informacemi jako je velikost, časové razítko a obsah jednotlivých paketů. Dále je dobré, aby uživatel věděl, jaké možnosti má při výběru sledované aplikace. Lze-li sledovat jednu aplikaci, více aplikací nebo veškerý provoz najednou. Pro pozdější zpracování je také nutné uložit soubor pcap pro zaznamenaný síťový tok. Posledním kritériem je, zda aplikace podporuje analýzu zaznamenaného síťového toku.

Aby bylo možné obsah paketů používajících protokol HTTPS dešifrovat, tak je nutné nainstalovat certifikát nabízený aplikací. Díky tomu lze provést útok typu *man in the middle* na danou komunikaci a obě komunikující strany jsou nuceny posílat své zprávy prostřednictvím útočníka. V tomto případě prostřednictvím monitorovací aplikace. Instalace uživatelských certifikátů prošla změnou v operačním systému Android N. Nicméně i u aplikací určených pro verze Android N a vyšší existují způsoby, jak nainstalovat takový certifikát. Pro provedení této operace je nutné nastavení důvěryhodné certifikační autority pomocí *Network Security Configuration*.

Existuje tedy několik dalších aplikací, které poskytují informace o obsahu odesílaných paketů. Avšak mají velkou nevýhodu ve způsobu, jakým zobrazují informace pro uživatele. To znamená, že přidaná hodnota mnou navrhované aplikace oproti ostatním na trhu je poskytnutí již konkrétně rozdělených informací, místo vypsání celého paketu, jak tomu je u ostatních aplikací. Poskytuje navíc analýzu zaznamenaného síťového provozu a možnost zaznamenávat více aplikací najednou, což u ostatních lze monitorovat jen jednu aplikaci nebo všechen provoz. Oproti tomu dvě ze tří testovaných aplikací umožňují nahlédnout do komunikace zabezpečené protokolem HTTPS, avšak mnou navrhovaná aplikace tuto funkcionalitu poskytovat nebude. Poslední obrovský rozdíl je v tom, že AppCheck nemá za cíl poskytovat výpis informací z přenesených paketů. Jejím cílem je poskytovat analytické informace o přenesených paketech namísto toho, aby v nich uživatel vyhledával.

[1] Viz https://play.google.com/store/apps/details?id=com.rinacode.android.netstatplus
[2] Viz https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture
[3] Viz https://play.google.com/store/apps/details?id=com.minhui.networkcapture

	NP	PC	NC	AC
UDP	A	N	A	A
TCP	A	A	A	A
IPv4	A	A	A	A
IPv6	A	A	A	A
Cílový port	A	A	A	A
Zdrojový port	N	N	N	A
Čitelný obsah HTTPS paketů	N	A	A	N
Informace o aplikaci	N	N	N	A
Zobrazení připojení	A	A	A	A
Stav připojení	A	N	N	A
Zachycuje poslané pakety	N	A	A	A
Zaznamenává velikost paketu	N	N	A	N
Časové razítko paketu	N	A	A	N
Informace o obsahu paketu	N	A	A	N
Zaznamenává komunikaci jedné aplikace	N	A	A	A
Zaznamenává komunikaci více aplikací	N	N	N	A
Zaznamenává komunikaci všech aplikací/veškerý provoz	N	A	A	A
Ukládá zaznamenaný provoz do pcap souboru	N	A	N	A
Analyzuje míru bezpečnosti aplikací	N	N	N	A

Tabulka 4.1: Tabulka srovnání jednotlivých aplikací vzhledem k jejich funkcionalitám. Názvy porovnávaných aplikací jsou zkráceny: NP = Netstat Plus, PC = Packet Capture, NC = NetCapture, AC = AppCheck. Zkratky hodnot tabulky: A =ANO, N = NE.

4.2 Získání procesů a síťových připojení

Pro získání informací ohledně síťových připojení a nainstalovaných aplikací bylo nutné využít poznatku, že jádro Androidu má základ v Linux kernelu (informace ohledně jádra Androidu jsou blíže popsány v kapitole 3.1.2). Na starších systémech, tj. Android 6 a nižší, bylo poměrně snadné získat dostatek informací ohledně systému. To se značně změnilo s verzí Android 7, kde se firma Google zaměřila hlavně na zvýšení bezpečnosti zařízení. Pro nalezení správného postupu, jak nutné informace získat, bylo potřebné provést několik experimentů.

Původní myšlenka byla získat přehled všech běžících procesů a na základě jejich *Process Identifier*[4] (PID) určit, které aplikace jsou aktuálně používané. Na nižších verzích se pro to využíval `ActivityManager`[5], konkrétně pak metoda `getRunningTasks()`, která je ve verzi Android 7 a výš zakázaná. Pro zpětnou kompatibilitu vrací pouze procesy patřící aplikaci, která si jejich seznam vyžádala. Toto řešení však nebylo dostatečné. Informace o aplikacích bylo nutné vyřešit jinak, konkrétně využitím třídy `PackageManager`[6]. Ta umožňuje za pomoci metody `getInstalledApplications()` získat údaje, jakými jsou *User ID*[7] (UID), název aplikace, její ikonku a verzi. Díky tomu lze zobrazovat seznam nainstalovaných aplikací. Problém s UID je obecně v tom, že oproti PID nemusí být unikátní. Během experimentu s touto funkcionalitou se nestalo, že by se vyskytla dvě stejná UID.

[4] PID je jedinečný identifikátor procesu přiřazený operačním systémem.
[5] Viz https://developer.android.com/reference/android/app/ActivityManager.html
[6] Viz https://developer.android.com/reference/android/content/pm/PackageManager
[7] *User ID*, zkráceně UID, identifikuje aplikaci v rámci operačního systému Android.

Při snaze získat PID jsem zkoušela využít i linuxového základu jádra Androidu. Podstatné pro tento experiment bylo zjistit možnosti provedení unixového příkazu v běhovém prostředí Androidu. Kvůli omezením, které jsou v jádře nastaveny, však nelze provést libovolný příkaz. Nicméně k provedení aspoň malé množiny příkazů lze použít třídu `Runtime`[8], která umožňuje provést některé unixové příkazy pomocí
`Runtime.getRuntime().exec('požadovaný příkaz')`. Bohužel se ukázalo, že příkazy týkající se procesů jsou buď nepodporované či zablokované. Závěrem je, že PID všech aplikací získat nelze.

Tímto experimentem se však ukázalo, že zavolání příkazu z běžící aplikace možné je. Zjistila jsem, že k získání informací o ustanovených síťových připojeních bude možné využít dva přístupy. První je pomocí nástroje nástroje *netstat*[9], který bez dodatečných parametrů vypisuje seznam otevřených unixových soketů a ustanovených internetových připojení pomocí protokolů TCP a UDP. Případně pomocí parsování souborů ze složky */proc/net/*, které obsahují obdobné informace a jsou rozděleny podle typu transportního protokolu i podle verzí IP protokolů na: *tcp*, *tcp6*, *udp* a *udp6*. Soubory označené číslem 6 obsahují informace o IPv6, soubory bez označení pak o IPv4. Oba způsoby poskytují informace o UID aplikace, k níž dané spojení náleží. Díky tomu lze všechna připojení namapovat k nainstalovaným aplikacím.

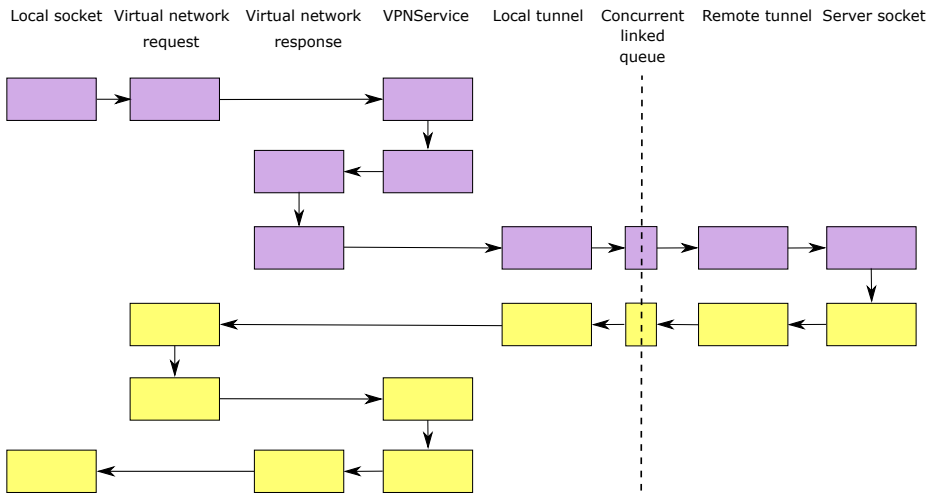
4.3 Zachytávání síťového provozu

Jádrem mé aplikace je práce se síťovým provozem mobilního zařízení. První fází zpracování je zachycení jednotlivých paketů při průchodu skrze síťové prvky. Pro tento účel je vhodná VPN, navíc ji lze implementovat pomocí třídy *VpnService* v balíčku *android.net*. Vytváří v aplikaci bod, skrze který prochází všechna data z/do mobilního zařízení. U odchozích a příchozích paketů je nutné zaznamenat jejich původní cíl a přesměrovat je na VPN, kde budou uloženy pro pozdější zpracování a následně přeposlány na jejich původní cílovou adresu.

Jako teoretický základ pro zachytávání paketů jsem použila hlavně dva články. První, nazvaný *Android VPN Service Explained with Packet Bypass Example Programme* [16], obsahuje příklad kódu, jak vytvořit VPN připojenou ke vzdálenému serveru, včetně slovního popisu v jaké části by měly být pakety zachytávány a jak s nimi pracovat. Tento článek mě inspiroval pro první návrh aplikace, ve kterém jsem využila VPN připojenou ke vzdálenému serveru. Po jeho dokončení jsem na základě zmíněného článku došla k názoru, že zachytávání paketů bez nutnosti využití vzdáleného serveru musí být proveditelné. Tj. že se musí dát realizovat zaznamenávání paketů jen v rámci mobilního zařízení. Avšak článek tuto možnost konkrétně neuvádí. Nicméně ve druhém článku [8] se pan Juàn Fēi Zhī Hái zaměřuje na bypass paketů v mobilním zařízení. Poskytuje přesnější informace o tom, jak implementovat uvedené řešení, jenž se stalo klíčovým pro zachycení paketů v navrhované aplikaci. Pokyny, týkající se zaznamenávání paketů, zmíněné v prvním článku [16] lze použít jak pro VPN připojenou ke vzdálenému serveru, tak při použití bypassu paketu.

Snaha najít lepší řešení zachytávání paketů, než za pomoci vzdáleného serveru, pramenila z nedostatečné kapacity a nízké úrovně zabezpečení mnou implementovaného serveru. Ten jsem vytvářela kvůli otestování prvního návrhu aplikace. Byla to poměrně jednoduchá implementace, která běžela na platformě Raspberry PI a neměla příliš dobrou konektivitu

[8] Dokumentace třídy `Runtime`: https://developer.android.com/reference/java/lang/Runtime
[9] Manuálová stránka pro příkaz *netstat*: https://linux.die.net/man/8/netstat



Obrázek 4.2: Vizualizace bypassu paketu skrze rozhraní navržené autorem článku [8]. Bloky znázorněné fialovou barvou jsou používány pro odchozí pakety. Žlutou barvou jsou znázorněny bloky pro příchozí pakety. Vlevo od šrafované čáry se nachází bloky používané pro manipulaci s pakety uvnitř mobilního zařízení. Oproti tomu bloky vpravo se využívají pro odesílání paketů na jejich původní cílovou adresu či pro jejich přijímání z původní zdrojové adresy.

ani kapacitu. Dalším problémem vzdáleného serveru je podle mého názoru nedůvěryhodnost. Z uživatelského hlediska nemám ráda, když aplikace shromažďuje mé údaje a odesílá je na externí server, o kterém nic nevím. Kvůli tomu jsem usoudila, že bude lepší najít jiné řešení, než se zabývat vylepšováním serveru.

Obrázek 4.2 ilustruje tunelování paketů pomocí rozhraní navrženého autorem druhého článku [8]. Implementační bloky, které se používají pro odesílání paketů z mobilního zařízení, mají fialovou barvu. Žluté se používají pro jejich příjem. Jednotlivé bloky používané pro manipulaci s pakety uvnitř mobilního zařízení lze vidět na obrázku oddělené šrafovanou čarou vlevo. Naopak, bloky používané pro odesílání dat do požadovaného cíle jsou umístěny vpravo. Výsledná navrhovaná aplikace používá téměř stejný bypass, jak je uvedeno v článku [8]. Díky tomu poskytuje bezpečný způsob, jak zachytit pakety a získat z nich informace, aniž by byly odesílány na vzdálený server. To eliminuje riziko, že by mohly být zachyceny neoprávněným uživatelem. Další výhodou je, že pro zpracování dat nebude využíván žádný externí server, který by pro uživatele mohl být nedůvěryhodný.

4.4 Ukládání paketů pro další zpracování

V předchozí kapitole je popsáno, v jaké části aplikace by měly být odesílané a přijímané pakety zaznamenávány. Způsobů, jak tyto data reprezentovat a ukládat je nespočet, avšak pro navrhovanou aplikaci jsem zvolila ukládání do souborů typu pcap. Jeho struktura je blíže popsána v kapitole 3.3. Cílem je, aby mnou implementovaný formát pcap souboru byl čitelný pomocí běžných nástrojů, jako je např. Wireshark.

Pro práci s těmito soubory existuje mnoho knihoven a nástrojů. Nicméně pro potřeby vytváření globálních a paketových hlaviček pcap souboru k zachyceným paketům jsem nenašla žádnou vhodnou knihovnu. Proto jsem navrhla *Pcap library*, která je umožňuje vytvořit. Globální hlavička souboru má pevně danou strukturu i obsah, proto i její implementace

bude jednotná pro všechny zaznamenávané síťové toky. Hlavička jednotlivých paketů je z tohoto pohledu komplikovanější. Pro její správnou implementaci je nutné znát celkovou velikost paketu včetně jeho L3 a L2 hlavičky. Ze zaznamenávaného toku lze získat pro jednotlivé pakety pouze L3 hlavičku spolu s jejich offsetem. Před uložením do souboru je tedy nutné získat i L2 hlavičku pro každý paket.

Pokud bych chtěla získat originální L2 hlavičku, přenesenou během síťové komunikace, tak by bylo nutné mít vyšší oprávnění než má běžný uživatel, tj. oprávnění uživatele *root* (viz kapitola 2.5.4). Avšak hlavička L2 nenese informace nezbytné pro správnou implementaci funkcionality aplikace. Z toho důvodu jsem se rozhodla hlavičku vytvořit jen pro forma a uložit do ní UID konkrétní aplikace, ke které se daný paket váže. Díky tomu bude možné v části analýzy zjistit, ke které konkrétní aplikaci pakety patří. Pro uložení UID jsem vybrala pozici zdrojové adresy, která by jinak nenesla hodnotnou informaci. Díky tomu zůstane zachovaný formát hlavičky a výsledný pcap bude korektní. Umístění UID v L2 hlavičce je zobrazeno ve výpisu 4.1, kde první uvedená ukázka je reálně využívaná L2 hlavička. Druhá nastiňuje umístění UID aplikace, které bude ukládané pomocí datového typu `Int`. Jedna MAC adresa má velikost 48b. Reprezentace celočíselného datového typu Int má v Androidu 32b, proto bude zapisována po 8b na pozice označené pomocí hodnot `UU`. Zbylé dvě pozice zdrojové MAC adresy budou vynulovány.

```
Běžná L2 hlavička:
  80  00  20  7A  3F  3E    80  00  20  20  3A  AE    80 00
   Cílová  MAC  adresa       Zdrojová MAC adresa    EtherType

Upravená L2 hlavička:
  80  00  20  7A  3F  3E    UU  UU  UU  UU  00  00    80 00
   Cílová  MAC  adresa       Zdrojová MAC adresa    EtherType
```

Výpis 4.1: Nástin využití hlavičky L2 pro uchování UID uvnitř pcap souboru v kontrastu k běžné L2 hlavičce. Pozice, značené v upravené adrese jako UU, obsahují UID konkrétní aplikace.

4.5 Analýza paketů

Po úspěšném zaznamenání síťového toku do pcap souborů je nutné začít řešit jejich analýzu. Je potřeba definovat, co bude považováno za bezpečné chování aplikace a co ne. Pro jednoduchost jsem se rozhodla nejprve rozdělit komunikaci na zabezpečenou, tj. tu která je uskutečněna pomocí protokolu HTTPS, a nezabezpečenou, tj. komunikaci realizovanou pomocí HTTP. I když jde u protokolu HTTP o nezabezpečenou komunikaci, nemusí to striktně znamenat, že se jedná o komunikaci nebezpečnou. Některá odesílaná data nemusí obsahovat citlivé údaje. Jedná se třeba o komunikaci na známý server či poskytování veřejně známých dat. Zda se jedná o komunikaci pomocí protokolu HTTP či HTTPS bude rozhodováno na základě využívaného portu. Port 80 je využíván pro HTTP. U HTTPS se používá port 443.

Samotnou HTTP komunikaci je vhodné rozdělit na bezpečnou a nebezpečnou. Pokud se posílají obecně známá data, tak se dá říct, že je komunikace bezpečná. Potencionální útočník se z této komunikace nedozví žádné klíčové informace. Jiná situace nastává, pokud se v komunikaci objeví citlivé údaje o aplikaci či uživateli. Může to být heslo uživatele, přístupové klíče aplikace a podobně závažné věci. Někdy se může stát,

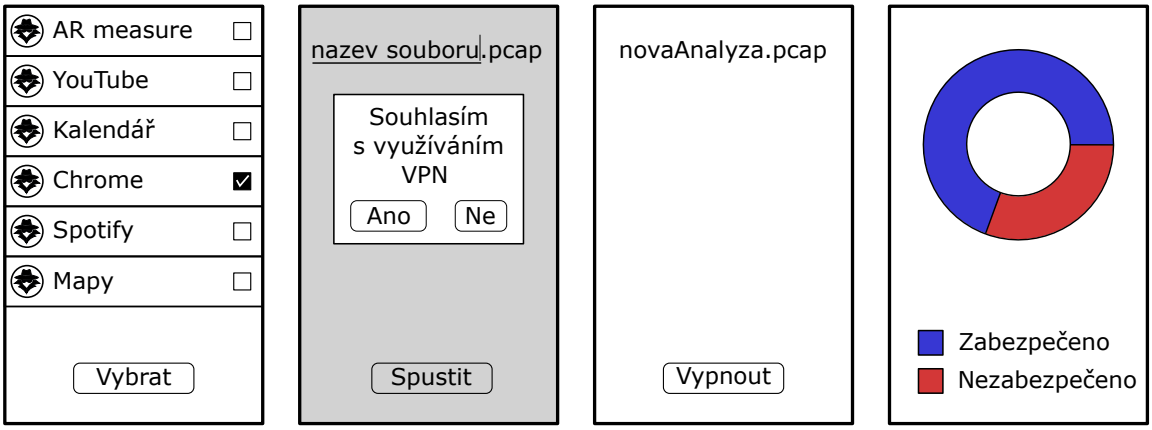
že návrhář aplikace zcela nerozumně použije pro poslání tohoto typu dat například tvar „secret=F6E69BD9DB9B0FD7E3152F36B9A4B891“, který byl již zmíněn v úvodu kapitoly, viz obrázek 4.1. Zde je využito klíčové slovo *secret*, jako identifikátor tajemství aplikace, avšak jeho určení tímto způsobem není pravidlem. Každá aplikace se chová jinak a odesílá jiná data. Díky tomu je poměrně složité algoritmicky zjistit, která informace může být citlivá. Proto se moje aplikace bude zaměřovat zejména na propojení monitorované aplikace, využitého protokolu, využité metody daného protokolu a URL adresy kam daný paket směřuje.

Data zaslaná pomocí HTTPS také nejsou úplně v bezpečí. Jako bezpečnostní prvek se zde využívá protokol SSL (viz kapitola 2.2.3) či TLS (viz kapitola 2.2.4). Útočník může na takto zabezpečenou komunikaci zaútočit pomocí *man in the middle* útoku (viz příloha A). Po jeho úspěšném provedení by se stal prostředníkem mezi dvěma legitimními koncovými body a dokázal by číst jejich vzájemnou komunikaci. Pokud by v takto zasílaných paketech byla citlivá data, tak by je získal. Nastíněný způsob je jeden z mála, který lze použít pro zjištění dat, která jsou v HTTPS komunikaci obsažena. Pokud by navrhovaná aplikace měla umět číst i tuto komunikaci, tak by bylo potřebné zmíněný útok realizovat. Avšak v rámci mobilního zařízení je tento útok poměrně komplikovaný. Bylo by potřebné vytvořit falešnou certifikační autoritu, jejíž certifikát by byl přidán mezi důvěryhodné v daném operačním systému. Pomocí ní by se podepisovaly další, dynamicky generované certifikáty, které by byly předávány zařízení oběti. Pomocí nich by šlo vytvořit důvěryhodné spojení a tak získat požadované informace. Tento způsob útoku byl popsán v článku *A study of SSL Proxy attacks on Android and iOS mobile applications* [7]. Nicméně aplikace se zaměřuje na naprosto nezabezpečená data, proto nebude tento přístup implementován.

4.6 Propojení paketů a aplikace

Propojení odesílaných a přijímaných paketů s konkrétní aplikací vychází z poznatků kapitol 4.2 a 4.4. Kvůli omezením, které přišly s Android 7, nelze získat informace o PID či UID pro jednotlivá síťová připojení získaná příkazem *netstat*. Ty by dopomohly k přesnému určení aplikace, která konkrétní spojení vytvořila. Navíc se při testování ukázalo, že se tento příkaz chová na různých implementacích operačního systému Android různě. Testovala jsem konkrétně implementace firmy LG[10], Gigabyte[11] a XiaoMi[12]. Kvůli tomu bude navrhovaná aplikace využívat informací získaných ze složky */proc/net/* z jádra operačního systému Android. Jsou v ní uloženy již zmiňované soubory *tcp*, *tcp6*, *udp* a *udp6*. Obsahují záznamy o síťových připojeních mobilního zařízení. V *tcp* jsou uchována spojení uskutečněná pomocí TCP protokolu a IPv4 protokolu, oproti tomu *tcp6* využívá TCP a IPv6. Obdobně je tomu u *udp* a *udp6*, kde je místo TCP využitý UDP protokol. Všechny soubory obsahují podobné informace. Pro navrhovanou aplikaci jsou důležité zejména zdrojová a cílová IP adresa, zdrojový a cílový port, stav spojení a především UID aplikace, která dané spojení vytvořila.

[10] Viz https://www.lg.com/
[11] Viz http://www.gigabyte.cz
[12] Viz https://www.xiaomi-czech.cz



Obrázek 4.3: Minimální návrh aplikace AppCheck, který pokrývá pouze základní funkcionalitu. Uživatel si může vybrat aplikaci, kterou chce monitorovat. Následně je potřeba, aby potvrdil, že souhlasí se spuštěním VPN. Pak může s vybranou aplikací pracovat mezitím co AppCheck bude na pozadí zachytávat její komunikaci. Posledním krokem je vypnutí VPN a získání analýzy míry bezpečnosti vybrané aplikace.

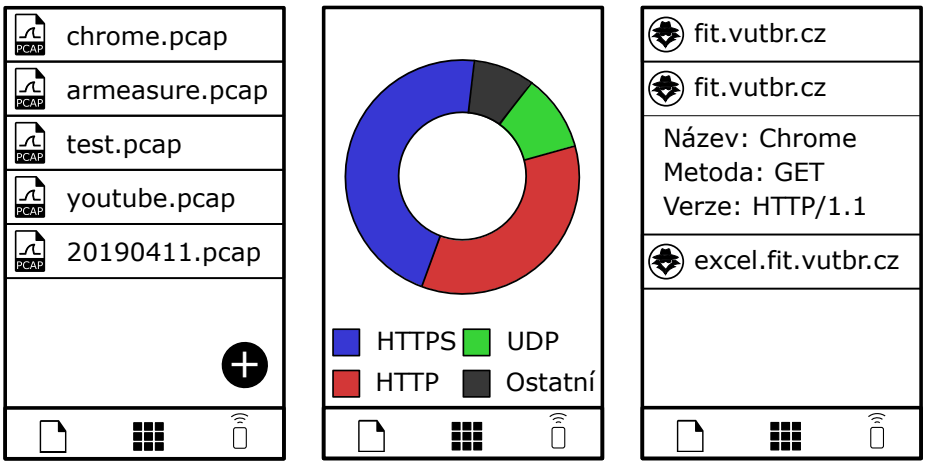
Nainstalované aplikace lze spojit s jejich síťovými připojeními pomocí zmíněných znalostí a informací získaných metodou `getInstalledApplications()` z třídy `PackageManager`, což bylo již zmíněno v kapitole 4.2. Následné propojení monitorované aplikace a paketů je nasnadě. Během zaznamenávání paketů, popsaného v kapitole 4.3, je možné získat zdrojovou i cílovou IP adresu a porovnat je s adresami, které se pojí ke konkrétní aplikaci. To je využito při rozhodování, zda paket patří některé z monitorovaných aplikací.

4.7 Návrh uživatelského rozhraní

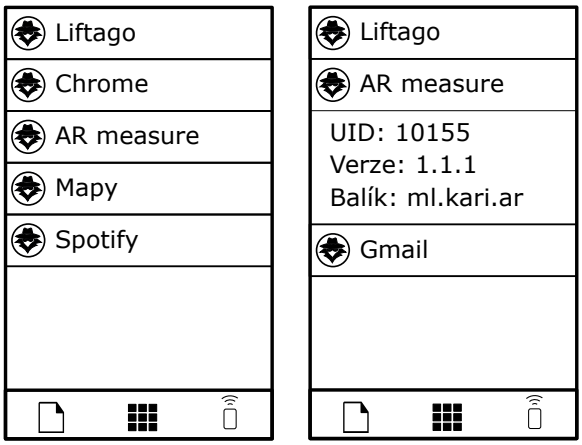
Jednotlivé komponenty uživatelského rozhraní jsem se snažila volit tak, aby se v nich běžný uživatel dokázal v krátkém čase zorientovat. Jako první jsem si vytyčila, jak by měly vypadat jednotlivé kroky mezi spuštěním aplikace a získáním analytiky týkající se uživatelem zvolené aplikace. Cílem bylo redukovat počet ovládacích prvků na minimální možnou úroveň, aby bylo ještě pochopitelné, co aplikace v daném kroku dělá. Na obrázku 4.3 je zachycena následnost jednotlivých obrazovek nutných pro zajištění funkcionality minimální verze. Vlevo je zobrazen seznam, ze kterého by si uživatel vybere jednu či více aplikací, které chce monitorovat. Poté je zobrazen systémový dialog, který žádá o povolení připojení k VPN. Po jeho potvrzení se zařízení uživatele připojí k VPN. Následně bude od uživatele požadována interakce s monitorovanou aplikací. Po určité době práce s ní se uživatel vrátí do AppCheck a vypnutím VPN získá analýzu míry bezpečnosti, což je naznačeno na obrázku vpravo.

Na základě zmíněného návrhu jsem po otestování minimálního produktu zpracovala návrh kompletního řešení se všemi požadovanými funkcionalitami. Chtěla jsem zakomponovat možnost zobrazení informací k nainstalovaným aplikacím, tj. jejich název, ikonku, UID a název balíku aplikace. U síťových připojení jejich zdrojovou a cílovou IP adresu, zdrojový a cílový port, verzi protokolu IP, typ připojení TCP či UPD, aktuální stav připojení, UID aplikace a její název.

Obrázek 4.4: Vytvoření nové analýzy síťového toku vybrané aplikace.



Obrázek 4.5: Zobrazení již zaznamenaného síťového toku včetně detailu.



Obrázek 4.6: Zobrazení informací o nainstalované aplikaci včetně detailního popisu.

Zvolila jsem rozložení aplikace pomocí *bottom navigation*, aby byly dílčí funkcionality aplikace snadno dostupné. Na obrázcích 4.4 až 4.6 jsou naznačeny tři případy užití, které mohou v aplikaci nastat. Slouží jako ukázka jednotlivých funkcionalit aplikace. První obrá-

zek ukazuje možnost, kdy chce uživatel vytvořit novou analýzu míry bezpečnosti konkrétní aplikace. Druhý obrázek demonstruje případ, kdy si uživatel chce prohlédnout již zaznamenanou analýzu včetně podrobných informací, které poskytuje. Třetí případ je zobrazení nainstalovaných aplikací včetně detailních informací. Obdobně by vypadal i čtvrtý případ užití, který není na obrázcích znázorněn. Týkal by se zobrazení aktivních síťových připojení, ke kterým jsou dostupné podrobné informace a díky tomu je svým vzhledem srovnatelný se třetím případem.

V případě obrazovek se základními informacemi a detaily, jako je naznačeno na obrázku 4.6, jsem se zaměřila na fakt, že uživatel pravděpodobně nepotřebuje vždy vidět detailní informace ke všem položkám. Proto jsem navrhla skrývatelný detail každé položky seznamu, který je dostupný kliknutím na položku aplikace či síťového připojení. Díky tomu si uživatel může rychle projít všechny nainstalované aplikace či ustanovená připojení, aniž by byl přehlcen informacemi. V případě zájmu o konkrétní položku se při jejím rozkliknutí dozví všechny dostupné informace.

4.8 Vylepšení aplikace na základě uživatelského testování

Kompletní návrh aplikace nastíněný v předchozí kapitole se osvědčil jako funkční základ designu. Nicméně během prvních několika testování se ukázalo, že nebyl nejvhodnější. Na základě poznatků z testování jsem ho iterativně vylepšovala a za pomoci několika testerů se mi podařilo podchytit většinu špatně rozmístěných komponent. Zejména u vytvoření nové analýzy uživatelé naráželi na problém nevhodně umístěného tlačítka pro potvrzení výběru aplikací pro monitorování. To bylo na základě připomínek zmenšeno a sladěno s designem aplikace.

Uživatelsky přívětivým krokem bylo také přidání možnosti smazat pcap soubor přímo v aplikaci. Když si uživatelé v první verzi aplikace vytvořili analýzu, kterou chtěli následně smazat, protože jim přišla nedostatečná nebo se jim během jejich testování odpojila VPN, tak museli tento soubor smazat v lokálním úložišti mobilního zařízení. Což je zdržovalo a většinou to vedlo k nepřehlednosti seznamu s uloženými pcap soubory. Kvůli tomu se v něm pak dalo stěží vyznat. Proto jsem v novém návrhu zakomponovala možnost soubor smazat dočasně či úplně. Druhým zajímavým požadavkem ze strany uživatele bylo přidat datum poslední aktualizace do detailu aplikace. Důvodem bylo, že je pak možné rychle zjistit, zda má danou aplikaci aktuální nebo existuje novější verze, kde mohla být potencionální chyba opravena. V neposlední řadě jsem řešila úpravy obrazovek, aby lépe vystihovaly svůj účel.

Doposud zmíněná vylepšení vznikla ve spolupráci s testery během vývoje aplikace. S již hotovou aplikací jsem se zúčastnila studentské konference Excel@FIT[13], na které jsem každého návštěvníka svého plakátu vybízela k otestování mé aplikace. Díky tomu jsem dostala podněty od odborné veřejnosti i nápady na vylepšení z řad akademických pracovníků a zástupců firem. Mezi ty nejpřínosnější patřilo vytvoření zjednodušené analytiky pro méně znalé uživatele či poskytnutí více dostupných informací v případě komunikace využívající jiného protokolu než je HTTP. Některá z navržených vylepšení jsem zapracovala do finálního návrhu aplikace.

[13] Viz http://excel.fit.vutbr.cz

Obrázek 4.7: Porovnání snímků původní verze (obrázky nahoře) a vylepšené verze (obrázky dole).

Pro srovnání původní verze a výsledné verze slouží obrázek 4.7. V horní části jsou umístěny obrazovky původní aplikace. Ty vznikly po vytvoření plnohodnotné implementace navržených funkcionalit. Jedná se o realizaci kompletního návrhu bez úprav navrhovaných ze strany testerů i návštěvníků konference. Ve spodní části jsou snímky obrazovky již vylepšené aplikace. Jedná se tedy o snímky z finální implementace aplikace AppCheck.

Kapitola 5

Implementace

Aplikace pro Android lze implementovat několika různými přístupy. Z pohledu architekturních vzorů jsou nejčastěji zmiňované přístupy MVC, MVP a MVVM, o kterých se zmíním později. Z pohledu návrhu lze vytvořit aplikaci několika způsoby. Například jen s využitím aktivit nebo s využitím kombinace aktivit a fragmentů. Druhý zmíněný způsob je častější, zejména díky možnosti upravovat vždy jen část obrazovky pomocí výměny či změny jednotlivých fragmentů. Speciálním případem je pak aplikace s architekturou *single activity*, kde existuje jen jedna aktivita a zbytek logiky je řešen pomocí fragmentů. Tu jsem vybrala pro svou aplikaci kvůli kompatibilitě s *bottom navigation*, jenž v aplikaci pohodlně zpřístupňuje všechny funkcionality. Navíc se architektura *single activity* stala podle článku[1] na *Android Developers Blog*, spravovaného firmou Google, preferovanou architekturou pro tvorbu Android aplikací.

Tato kapitola bude věnována implementaci jednotlivých částí aplikace. Letmo se zaměří na popis architekturních vzorů a bude konkretizovat ten, který byl vybrán pro vývoj aplikace AppCheck. Dále popíše architekturu *single activity* a jak vypadá její zakomponování do výsledné aplikace. A v neposlední řadě se bude zabývat způsobem implementace bypassu paketů za pomoci VPN rozhraní včetně problematiky extrakce dat ze síťového provozu.

5.1 Architekturní vzory

Existují tři základní architekturní vzory pro návrh Android aplikací. Jsou to již zmiňované MVC, MVP a MVVM. Jejich rozdíl spočívá v dělbě povinností mezi jednotlivé komponenty. *Model View Controller* je nejstarším ze jmenovaných vzorů. Funkcionalitu rozděluje mezi 3 komponenty. Konkrétně se jedná o:

- *Model* – ten zajišťuje práci s daty, jejich stavem a business logikou,
- *View* – zajišťuje reprezentaci modelu, má za úkol řešit renderování uživatelského rozhraní (UI) a komunikaci s *Controller* kdykoliv uživatel interaguje s aplikací a
- *Controler* – který je spojnicí mezi *Modelem* a *View*, tzn. kdykoliv uživatel provede nějakou akci v UI tak *Controller* vyhodnotí co se má stát na základě akce uživatele a informací uložených v *Modelu*, stejně jako když se změní informace v *Modelu*, tak je adekvátně zobrazí uživateli pomocí *View* do UI.

[1] Viz sekce *Navigation*: https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html?m=1

Hlavním problémem tohoto architekturního vzoru je jeho modularita a flexibilita. Kdykoliv když se bude měnit i malá část UI, tak je nutné zasahovat i do *Controller* komponenty, která může nabobtnat a stát se značně nepřehlednou.

Druhý je *Model View Presenter*. Ten přejímá část rozdělení podle MVC avšak zodpovědnost jeho největší komponenty *Controller* rozdělil. Obsahuje tedy:

- *Model* – má stejnou funkncionalitu jako u MVC,
- *View* – funguje obdobně jako u MVC, navíc ale obsahuje i aktivity a fragmenty, u nichž je vhodné implementovat rozhraní pro práci s nimi, aby mohl *Presenter* přistupovat ke kterékoliv aktivitě či fragmentu jednotně a
- *Presenter* – zajišťuje řízení aplikace na základě interakce uživatele a stavu modelu, avšak není připojen přímo k *View*, ale pracuje s ním pouze prostřednictvím jeho rozhraní.

Tento přístup umožňuje pohodlnější tvorbu *unit* testů. A je značně flexibilnější než MVC, zejména díky využití rozhraní pro práci s *View*.

Posledním z nich je *Model View ViewModel*, který vychází obdobně jako MVP z MVC avšak pojímá komponenty odlišně. Obsahuje:

- *Model* – srovnatelný s MVC i MVP,
- *View* – také se stará o UI, avšak změny se provádí na základě pozorování jednotlivých proměnných ve *ViewModel* komponentě, tj. za využití návrhového vzoru *observer*[2] a
- *ViewModel* – zastává roli prostředníka mezi *Model* a *View*, má na starosti bussiness logiku a přípravu dat pro zobrazení v UI, což řeší pomocí nastavování svých proměnných, které pak při změně upozorní *View*, že je potřeba překreslit UI.

MVVM je nejvhodnějším architekturním vzorem pro *unit* testování, protože *View* a zbytek logiky na sobě vůbec nezávisí. Jediné, co je potřebné, je emitovat změny hodnot pozorovaných proměnných, na jejichž základě se jednotlivé obrazovky překreslují.

Pro implementaci navržené aplikace AppCheck jsem vybrala právě vzor MVVM spolu s *Data Binding Library*[3] pro předávání informací mezi komponentami *View* a *ViewModel* za pomoci pozorovaných proměnných. Důvodem, proč jsem si ho vybrala, bylo jeho přizpůsobení pro *unit* testy a v neposlední řadě jeho, dle mého názoru, pochopitelná struktura.

5.2 Navigace a architektura Single Activity

Jak bylo již zmíněno v úvodu kapitoly, tak *single activity* architektura je vhodnou volbou pro vytvoření aplikace. Umožňuje kooperaci s *bottom navigation* díky architekturní komponentě *Navigation* z kolekce Android Jetpack. Ta je v mé implementaci řešena pomocí souboru `app_navigation.xml`, jenž je zařazen do balíčku `res` (*resources*). Z tohoto souboru se generuje tzv. graf navigace, který definuje, jak jsou jednotlivé fragmenty vzájemně provázány. Je potřeba si uvědomit, že celá aplikace pracuje jen s jednou hlavní aktivitou. V rámci jejího rozložení je definován jediný element fragmentu. Ten je typu `androidx.navigation.fragment.NavHostFragment` a má přiřazený soubor s grafem navigace, který se v rámci elementu fragmentu v rozložení definuje jako

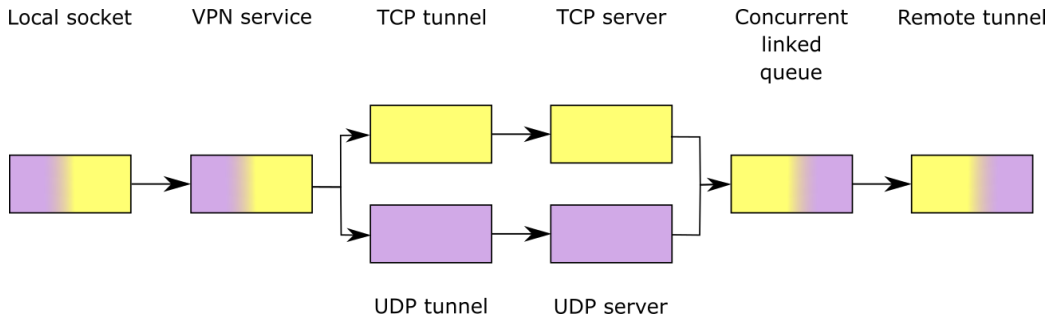
[2] Viz https://www.thedroidsonroids.com/blog/design-patterns-android-observer-2
[3] Viz https://developer.android.com/topic/libraries/data-binding

`app:navGraph="@navigation/app_navigation"`. Mimo to obsahuje i element `com.google.android.material.bottomnavigation.BottomNavigationView`, který zobrazuje panel pro *bottom navigation*, jenž je viditelný napříč všemi obrazovkami aplikace.

Navigaci mezi obrazovkami je nutné definovat v souboru `app_navigation.xml` pro každý fragment obdobně, jako je to naznačeno ve výpisu 5.1. Element `<fragment>` určuje, o který konkrétní fragment se v aplikaci jedná. V ukázce je zobrazena definice pro `AnalyzeFragment`, který definuje obrazovku se spouštěním VPN. Jde tedy o část, kde si uživatel může spustit VPN a zahájit tak monitorování komunikace požadované aplikace (či více aplikací). Ke každému fragmentu se ještě může pojit akce, definovaná pomocí elementu `<action>`. Ta specifikuje, na kterou obrazovku bude možné z daného fragmentu přejít. V tomto případě se jedná o akci `action_analyze_end`, která zobrazí graf analýzy zaznamenaného síťového toku. Dále je možné definovat argumenty jednotlivých fragmentů, které se určují pomocí elementu `<argument>` a mohou být jen základních datových typů. Proto je vhodné dobře promyslet, jaká data je nutné mezi obrazovkami předávat již při tvorbě návrhu. V zobrazeném příkladu se posílá mezi obrazovkou výběru aplikací a obrazovkou s možností zapnutí VPN seznam UID. Ten konkrétně určuje, které aplikace mají být monitorovány. Argument je tedy seznam celočíselných hodnot (`integer`), který reprezentuje vybrané aplikace.

```
<fragment
 android:id="@+id/nav_analyze"
 android:name="cz.blackchameleon.appcheck.fragment.AnalyzeFragment"
 android:label="@string/vpn_header"
 tools:layout="@layout/fragment_analyze"
 >
 <argument
  android:name="app"
  appDetail:argType="integer[]"
  />
 <action
  android:id="@+id/action_analyze_end"
  appDetail:destination="@id/nav_file_detail"
  appDetail:popUpTo="@id/nav_file_list"
  />
</fragment>
```

Výpis 5.1: Ukázka jednoho fragmentu v souboru `app_navigation.xml` pro definici navigace v aplikaci AppCheck. Zobrazená ukázka je včetně demonstrace akce a argumentu, které se k danému fragmentu vážou.



Obrázek 5.1: Schéma řešení bypassu paketů odchozího síťového toku v aplikaci AppCheck. Žlutou barvou jsou označeny pakety posílané pomocí protokolu TCP. Oproti tomu fialově jsou označeny pakety posílané pomocí protokolu UDP. Bloky, které jsou označeny oběma barvami, slouží pro bypass veškerých paketů. Zaznamenávání síťového toku do pcap souborů probíhá v bloku *VPN service*.

5.3 Bypass paketů

Jak již bylo řečeno v kapitole 4.3, tak samotnou implementaci VPN lze řešit dvěma způsoby. První je vytvořit pomocí VPN spojení na vzdálený server, skrze který budou posílány jednotlivé odchozí i příchozí pakety. Druhou možností je bypass paketů. V tomto případě se vytvořená VPN připojí pouze k virtuálnímu rozhraní vytvořenému v mobilním zařízení. Díky čemuž se pakety nebudou odesílat na vzdálený server a jejich zpracování proběhne jen v rámci mobilního zařízení.

Ukázku kódu pomocí kterého lze vytvořit VPN rozhraní nabízí i Google s jeho ToyVPN, což je zmíněno již v kapitole 2.5.1. S využitím této ukázky a části komentovaného kódu v článku pana Suna [16] jsem vytvořila vlastní implementaci VPN rozhraní. To slouží jako základ pro bypass paketů, který je z teoretického hlediska nastíněn v kapitole 4.3 a vychází z článku pana Juàn Fēi Zhī Háie [8]. Kvůli znovupoužitelnosti řešení a nemalému rozsahu implementace jsem vytvořila knihovnu *Bypass Packet Capture*, jenž obsahuje všechny nezbytné součásti pro bypass paketů v rámci mobilního zařízení s operačním systémem Android.

V knihovně je implementováno několik ucelených bloků potřebných pro zpracování paketů. Prvním je VPN rozhraní, které přeposílá lokálně vytvářené pakety do lokálního tunelu. Je implementovaná ve třídě `BypassPacketService`, která dědí od Android třídy *VpnService*[4]. Lokální tunel je rozdělen na dva typy podle použitého transportního protokolu, tj. TCP či UDP. Každý z nich je propojen s odpovídajícím serverem, který si na tento tunel uchovává referenci. V implementaci jsou pojmenovány jako `TcpServer` a `UdpServer`. Na ně navazuje fronta požadavků reprezentovaná pomocí instance třídy `ConcurrentLinkedQueue`. Jednotlivé pakety uložené ve frontě požadavků jsou postupně odesílány na vzdálený tunel implementovaný třídou `RemoteTunnel`. Ten slouží jako vstupní či výstupní bod celého zpracování paketů a je propojen s Internetem.

Obrázek 5.1 zobrazuje tok odchozích paketů, tj. těch paketů které směřují z mobilního zařízení do Internetu, skrze bloky pro bypass paketů. První blok, ve kterém se vytváří lokální paket je VPN. Na základě transportního protokolu zaznamenaného v jeho hlavičce se rozhodne, který lokální tunel se použije. Než je paket přeposlán skrze TCP či UDP tunel, tak se k jeho L3 hlavičce připojuje L2 hlavička a hlavička paketu nezbytná pro uložení do

[4]Viz https://developer.android.com/reference/android/net/VpnService

pcap souboru. Nicméně nově připojená L2 hlavička nekoresponduje s původní přenášenou hlavičkou. Důvodem je to, že původní přenášenou hlavičku již paket během průchodu skrze VPN rozhraní neobsahuje. A protože v ní nejsou obsaženy žádné pro aplikaci nezbytné informace, tak jsem ji využila jako úložiště pro konkrétní UID monitorované aplikace, jak je popsáno v kapitole 4.4. To znamená, že zaznamenání a uložení paketů do pcap souboru je součástí bloku *VPN service*. Následuje poslání paketu skrze určený tunel na příslušný server. Celý proces je implementován tak, aby ke každému paketu uchovával zdrojovou i cílovou IP adresu a zdrojový i cílový port. Nakonec se paket ze vzdáleného tunelu odešle do původního cíle.

5.4 Analýza zachycených paketů

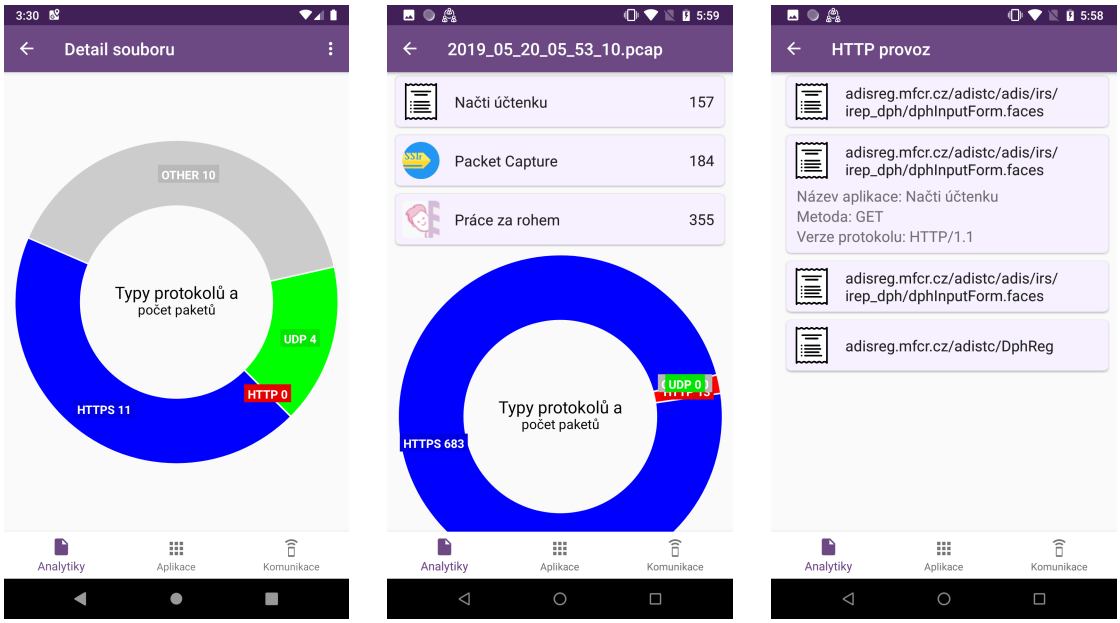
Získávání jednotlivých paketů ze síťového toku je implementačně řešeno v předcházející kapitole. Jak již bylo zmíněno, tak práce s hlavičkami jednotlivých paketů a jejich následné ukládání do pcap souborů se kvůli omezeným možnostem VPN rozhraní na Androidu částečně komplikuje. Proto jsem implementovala knihovnu *Pcap library*[5]. Obsahuje metody potřebné pro vytvoření pcap souborů z konkrétního *byte streamu*, který je získáván z VPN rozhraní v nekonečném cyklu. Struktura pcap souboru je popsána v kapitole 3.3. Před zahájením přenosu paketů je v lokálním úložišti vytvořen pcap soubor obsahující pouze globální hlavičku. V každé iteraci se pomocí této knihovny vytvoří paketová hlavička a připojí se k zachycenému paketu. Poté se vytvoří i nová L2 hlavička. Ta má místo zdrojové MAC adresy uložené UID monitorované aplikace, ke které daný paket náleží. V závěru iterace se L2 hlavička připojí k již hotovému bloku a celý se uloží do pcap souboru.

Pro vytvoření analýzy zaznamenaného síťového toku z uloženého souboru je nutné zpracovat postupně všechny pakety. Na to používám knihovnu *Okio*[6]. Ta poskytuje funkce pro zjištění protokolu paketu, cílového portu, zdrojového portu a mnoha dalších informací. Výsledkem by měla být grafová reprezentace počtu zaznamenaných paketů rozdělená podle rozlišovaných protokolů. Zpracování grafu probíhá v souboru `FileDetailViewModel` v metodě `showAnalyticsDetail()`. Zde je vytvořen *file stream* z požadovaného souboru, kde se ve smyčce analyzují jednotlivé pakety. Pro každý rozlišovaný protokol je vytvořen čítač, který je inkrementován na základě detekce paketu daného protokolu. První rozdělení je dle transportního protokolu. V případě rozpoznání UDP protokolu se paket dále nerozlišuje. Je-li transportním protokolem TCP, pak je u paketu porovnáván zdrojový a cílový port. Je-li aspoň jeden z nich roven číslu 80, tj. HTTP port, pak je inkrementován čítač HTTP protokolu. Má-li některý z nich číslo 443, pak se jedná o protokol HTTPS. Ty, které nevyhovují zadaným kritériím, jsou označeny jako „ostatní“.

Výsledný graf je vytvářen pomocí knihovny *HelloCharts for Android*[7]. Implementaci koláčového grafu ve zmíněné knihovně bylo nutné upravit tak, aby využívaný `onValueTouchListener` třídy `PieChartOnValueSelectListener` dokázal kromě zachycení gesta uživatele a místa jeho dotyku ještě otevřít novou obrazovku. Implementovaný graf je rozdělen podle rozlišovaných protokolů a velikost jednotlivých ploch závisí na hodnotě čítačů, jak je vidět na obrázku 5.2. Detail komunikace, na který se uživatel dostane kliknutím na požadovanou část grafu, se liší v závislosti na konkrétním protokolu. Pro HTTP obsahuje seznam jednotlivých URL, se kterými monitorovaná aplikace komunikovala. U ostatních typů ji nelze zjistit a proto jsou vypsána alespoň data, která jsou z paketu zjistitelná.

[5] *Pcap library* je veřejně dostupná na stránce https://github.com/klepackovakarolina/pcaplibrary
[6] Dostupná na adrese https://github.com/square/okio
[7] Viz https://github.com/lecho/hellocharts-android

Současně je vždy určeno, která z monitorovaných aplikací konkrétní požadavek odeslala. Na zmiňovaném obrázku vlevo je vidět, že při zaznamenávání komunikace bylo odesláno pomocí protokolu HTTP 0, HTTPS 11, UDP 4 a ostatními protokoly 10 paketů. Tato analýza byla vytvořena sledováním toku aplikace Liftago[8] po dobu 2 minut. Zbylé dva snímky, které spolu souvisí, demonstrují analýzu zaznamenaných paketů včetně jejich obsahu.



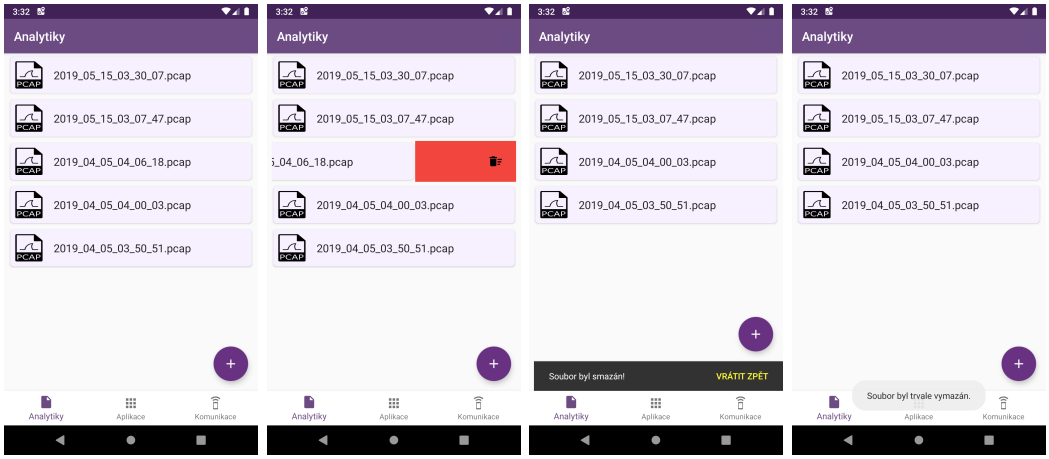
Obrázek 5.2: Graf vlevo zobrazuje zaznamenanou analýzu aplikace Liftago, která byla uskutečněna původní verzí aplikace. Zbylé dva obrázky na sebe navazují a ukazují zaznamenání síťového toku pro tři aplikace. Kliknutím na graf se zobrazí požadovaný detail, který je vidět na pravém obrázku.

5.5 Mazání souborů v aplikaci

Mazání pcap souborů v seznamu na hlavní stránce aplikace jsem řešila pomocí implementace třídy `SwipeToDeleteCallback`, která dědí od třídy `ItemTouchHelper.SimpleCallback`. Zaznamenává *swipe* pohyb uživatele na konkrétní pozici seznamu. Pokud uživatel udělá *swipe* gesto doleva, tak je to detekováno jako snaha o vymazání souboru. Uživateli se při částečném posunu položky zobrazí červený podklad spolu s ikonkou koše. Pokud je položka přetažena úplně doleva, pak je smazána. U této funkcionality jsem řešila zejména případ, kdy si uživatel smazání položky rozmyslí, případně ji smaže omylem. Vymyslela jsem poměrně snadno implementovatelný způsob, který zabrání nejen okamžitému smazání souboru, ale i zbytečnému obtěžování uživatele pomocí vyskakovacího dialogu. Tento způsob je ukázán pomocí obrázku 5.3. Uživatel nejdříve začne s pohybem vlevo, čili začne akci mazání souboru. Následně je soubor smazán a odebrán ze seznamu souborů. Uživateli v tu chvíli vyskočí *Snackbar*[9], který mu umožní danou akci vrátit zpět. Pokud je rozhodnutý soubor trvale smazat, pak nemusí nic dělat. Avšak chce-li smazání zabránit, pak klikne na tlačítko „Vrátit zpět“ a soubor se zpět objeví na původní pozici. Implementace tohoto problému

[8] Viz https://play.google.com/store/apps/details?id=com.adleritech.app.liftago.passenger
[9] Viz https://developer.android.com/reference/android/support/design/widget/Snackbar

je řešena jen dočasným skrytím souboru ze seznamu. Definitivně se soubor smaže až když zmizí *Snackbar* ve spodní části obrazovky. Proto bude soubor trvale smazán po uplynutí doby zobrazení, přejitím na jinou obrazovku, případně přepnutím aplikace na pozadí.



Obrázek 5.3: Zobrazení návaznosti akcí při mazání pcap souboru ze seznamu.

5.6 Výpočetně náročné operace

Pro implementaci výpočetně náročných operací jsem zvolila knihovny *RxJava*[10] a *RxKotlin*[11], které jsou součástí aplikačního rozhraní *ReactiveX*[12]. Obě umožňují asynchronní programování a jejich hlavním rozdílem je podporovaný programovací jazyk. Za správný přístup k výpočetně náročným operacím se na Androidu považuje zpracování na samostatných vláknech. Ve své aplikaci využívám tři typy vláken: `computation`, `io` a `mainThread`. První dvě zmíněné patří do knihovny *Schedulers*[13] a poslední patří do *AndroidSchedulers*[14].

Každé z vláken má určeno, pro jaké operace by se mělo používat. Vlákno `io` by se mělo využívat pro jakékoliv vstupně výstupní operace. Bývají to typicky dotazy na server či práce se souborovým systémem. Při využití tohoto vlákna dojde nejprve k vytvoření *worker* objektu, tj. vlákna které zodpovídá za řízení operace. U dlouho běžících operací si vytváří další podpůrná vlákna (využívají se jen pro zpracování krátkodobé činnosti) či *worker* objekty (které mohou dále proces dělit), aby dokázal pokrýt všechny požadavky na zpracování. Obdobné využití má i `computation`, avšak na rozdíl od předchozího typu se dá určit maximální počet vláken, která mohou být aktivní. Díky tomu nedochází ke snížení výpočetního výkonu vinou nadměrného počtu vláken. Posledním typem je `mainThread`, které je hlavním vláknem aplikace a proto by mělo být vytěžováno jen méně náročnými operacemi. Využívá se zejména na úpravu UI, což znamená, že jeho nadměrné využívání by způsobovalo „zamrzání“ aplikace, čemuž je lepší se vyhnout.

Komunikace vláken je založena na návrhovém vzoru *observer*. Ten využívá pojmů *subject* pro pozorovaný objekt a *observer* pro objekt pozorovatele. Mezi nimi platí vztah $1 \ldots N$, což znamená, že jeden *subject* může být sledován až N objekty typu *observer*. Vždy, když *subject* emituje novou hodnotu, tak se to dozví *observer* a obdrženou informaci může dále zpracovávat. Základními prvky řetězce zpracování asynchronní operace jsou v *ReactiveX*

[10] Viz https://github.com/ReactiveX/RxJava
[12] Viz http://reactivex.io
[13] Viz http://reactivex.io/RxJava/javadoc/rx/schedulers/Schedulers.html
[14] Viz https://github.com/ReactiveX/RxAndroid

API: *Observable* (zdroj emitovaných hodnot), *Operátor* (těch může být nad daným zdrojem $0 \ldots N$) a *Subscriber* (odebírá emitované položky). Rozšíření funkcionality výpočetních řetězců spočívá v množství využitých operátorů[15]. Jako ukázka poslouží kód 5.2, který je v aplikaci AppCheck využíván pro obnovování seznamu ustálených připojení v intervalu $2\,s$. Prvním řádkem se vytvoří zdroj emitovaných hodnot, který má provést akci každé $2\,s$. Druhý řádek specifikuje, že výpočet bude prováděn na `io` vlákně. Následně je potřebné označit, kam se budou vypočítané hodnoty emitovat, což je řečeno pomocí operace `observeOn` na třetím řádku. Zde je potřebné uvědomit právě `mainThread`, aby po získání nových informací došlo k překreslení UI. Poslední řádek definuje konkrétní kód, který bude asynchronně zpracován. Po dokončení výpočtu metody `requestCurrentConnectionsList()` se v každé iteraci emituje nová hodnota na `mainThread` a uživateli se zobrazí aktuální přehled síťových připojení.

```
1  Observable.interval(1000, 2000, TimeUnit.MILLISECONDS)
2    .subscribeOn(Schedulers.io())
3    .observeOn(AndroidSchedulers.mainThread())
4    .subscribe { requestCurrentConnectionsList() }
5
```

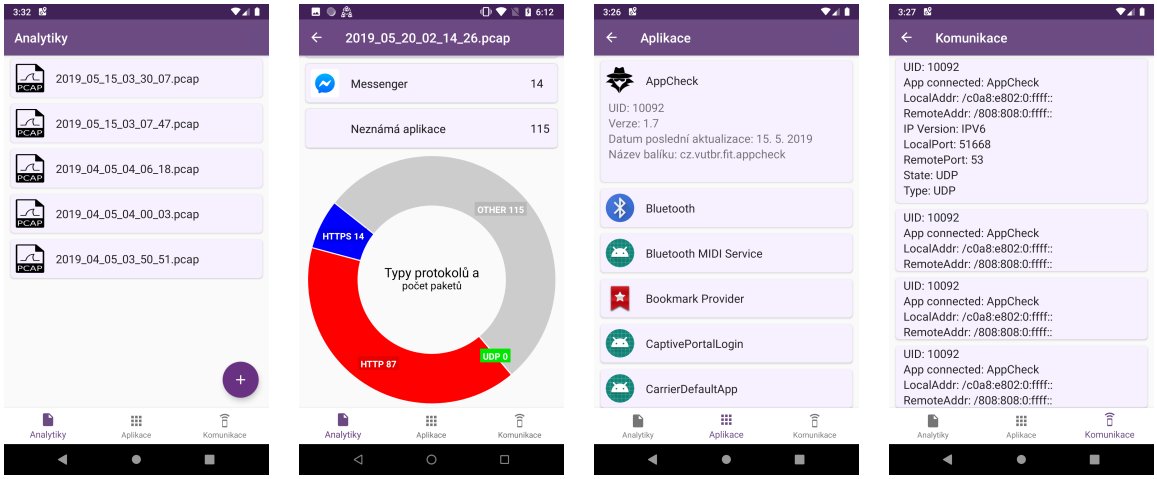
Výpis 5.2: Ukázka asynchronní operace pomocí knihovny *RxJava*.

[15] Viz http://reactivex.io/documentation/operators.html

Kapitola 6

Vyhodnocení dosažených cílů

Výsledná aplikace AppCheck[1] je plně funkční a dostupná k volnému stažení v obchodě Google Play. Je vhodná především pro uživatele, kteří mají povědomí o fungování internetových sítí a jejich potencionálním riziku. Umožňuje zaznamenat síťový tok monitorované aplikace (či více aplikací) pomocí VPN, který je následně uložen do pcap souboru. Zachycená data jsou využita pro analýzu míry bezpečnosti. S její pomocí lze odhalit aplikace, které komunikují nezabezpečeně pomocí protokolu HTTP. V případě zaslání paketu tímto protokolem umožňuje aplikace zjistit, jaké informace byly poslány v jeho hlavičce. Navíc poskytuje informace týkající se nainstalovaných aplikací. To zahrnuje informace o názvu, ikonce, verzi, datu poslední aktualizace, UID a jméně balíčku, který aplikaci obsahuje. Podobným principem jsou zpracována i aktivní síťová připojení. Informace k nim poskytnuté jsou název aplikace, která připojení vytvořila, její UID, zdrojová IP adresa, cílová IP adresa, zdrojový port, cílový port, verze transportního protokolu a aktuální stav připojení. Veškerou svou funkcionalitu poskytuje bez nutnosti využití oprávnění uživatele *root*.



Obrázek 6.1: Snímky hlavních obrazovek výsledné aplikace AppCheck. Vlevo je seznam zaznamenaných pcap souborů. Druhý obrázek zachycuje výsledný graf analýzy zaznamenaného síťového toku. Třetí zobrazuje aktuálně nainstalované aplikace, konkrétně je vidět detailní popis aplikace AppCheck. Na posledním snímku jsou informace o jednotlivých síťových připojeních.

[1]Dostupná na Google Play: https://play.google.com/store/apps/details?id=cz.vutbr.fit.appcheck

Na obrázku 6.1 jsou snímky výsledné aplikace. Jejím hlavním rozdílem oproti konkurenci je zejména graf míry bezpečnosti monitorované aplikace (či aplikací), který se nachází na druhém snímku zleva. Navíc dokáže u HTTP provozu určit obsah hlaviček jednotlivých paketů, aniž by uživatel musel tyto informace hledat sám ve výpisu celého paketu, jako je tomu u některých ostatních aplikací (viz srovnání v kapitole 4.1). Poskytnutí čitelného výpisu celého obsahu paketu je dle mého názoru spíše expertní znalost a proto bych ji ve své aplikaci poskytovala až v případě *expert* módu, který je blíže popsán v kapitole týkající se možných vylepšení 6.1.

Funkcionalita aplikace byla iterativně testována a vylepšována s každou verzí. Proběhlo celkem 6 iterací, kdy byli zapojeni nezávislí testeři. Z každého testování vzešly nové nápady na vylepšení, které byly postupně zakomponovány do výsledné verze. Kromě manuálního testování obsahuje implementace aplikace i několik *unit* testů, které se zaměřují především na otestování správnosti chování modelových tříd, včetně jejich metod.

6.1 Možná vylepšení

Mezi možná vylepšení stávající aplikace AppCheck patří v první řadě možnost získat obsah HTTPS paketů a zjistit, zda v nich nejsou odesílány citlivé údaje. Způsob, jakým lze tato data získat, byl nastíněn již v kapitole 4.1 a konkrétně popsán v kapitole 4.5.

Další změnou, která by mohla být pro uživatele přínosná by bylo vytvoření tří módů aplikace. První by poskytoval informace, které by byly uchopitelné pro nejméně zkušené uživatele. Aplikace by v módu *začátečník* určovala pouze míru zabezpečeně poslaných paketů, které by nebyly děleny na jednotlivé protokoly. Ve výpisu nainstalovaných aplikací a síťová připojení by poskytovala jen částečné informace. V módu *pokročilý* by se uživatel z analýzy dozvěděl počty paketů zaslaných konkrétními protokoly. U aplikací a síťových připojení by byly dostupné obdobné informace, jaké jsou nyní v AppCheck poskytovány. Celkově by se mód *pokročilý* nejvíce blížil současné implementaci. Posledním by byl *expert*, kde by bylo možné rozvést informace získané ze síťového toku podrobněji. Mohly by být přidány informace o velikosti jednotlivých paketů a jejich obsah. Také by se daly implementovat funkcionality, které by vyžadovaly oprávnění uživatele *root*. V módu *expert* by měl uživatel na výběr, zda tyto funkcionality povolí nebo ne. Přehled předpokládaných funkcionalit rozdělených pro jednotlivé módy je shrnutý v tabulce 6.2.

6.2 Nedostatky aplikace

Aplikace začala vznikat dříve, než byl oficiálně oznámen seznam změn, které budou z bezpečnostního hlediska provedeny na Androidu Q. Konkrétní část změn, která se přímo dotýká funkcionality aplikace je omezení přístupu ke složce */proc/net*, jejíž využití bylo zmíněno v návrhu aplikace v kapitole 4.6. Nově by mělo přibýt oprávnění, které bude nutné pro získání přístupu k obsahu této složky. Změny se chystají i pro využívání VPN. Konkrétně by se mělo jednat o propojení `NetworkStatsManager` a `ConnectivityManager` s implementovanou VPN. Tyto informace byly převzány ze stránky *Help Net Security*[2]. Díky těmto změnám by bylo nutné aplikaci AppCheck upravit podle novějších požadavků, avšak v době zpracování této diplomové práce ještě není k dispozici žádné zařízení, které by mělo operační systém Android Q. Proto není možné implementovanou aplikaci pro novou verzi Androidu přizpůsobit.

[2]Viz https://www.helpnetsecurity.com/2019/03/18/android-q-privacy/

	HTTP	HTTPS	UDP	ostatní
9gag	129	1773	223	4128
Chrome (stránka Excel@fit)	18	240	63	375
Slack	0	850	32	874
AR measure	0	137	12	160
Remote desktop	31	37	467	931
IDOS	2	616	381	670
World Newspapers	807	482	482	1467
Network Scanner	7	66	19	73

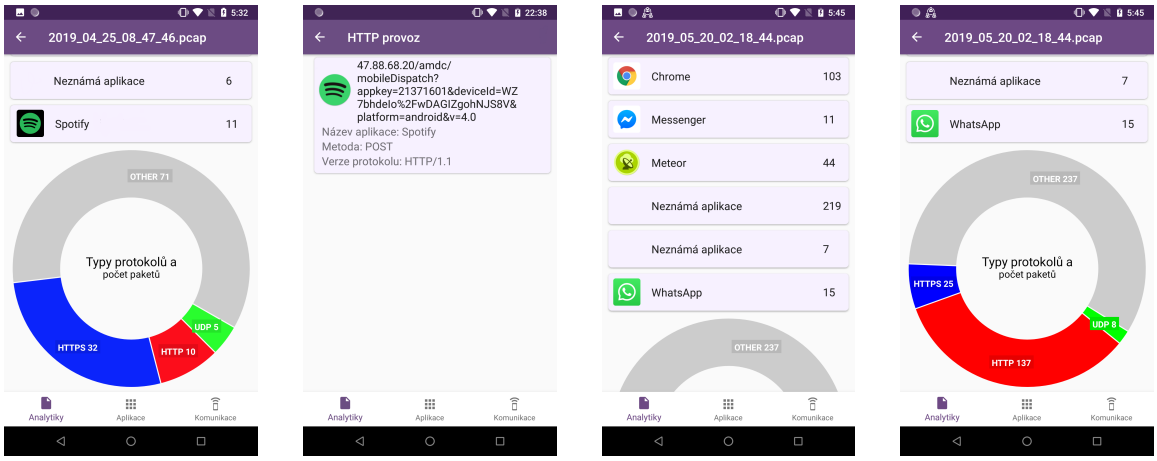
Tabulka 6.1: Analýza jednotlivých aplikací. Jsou zde zaznamenány počty paketů, které byly poslány pomocí konkrétních protokolů. Každá analýza byla uskutečněna po zaznamenání síťového toku po dobu cca 1 minuty.

6.3 Výsledné statistiky

Pro otestování výsledné aplikace jsem provedla několik analýz. Jednou z nich byl test funkcionality na aplikacích: 9gag[3], Chrome[4], Slack[5], AR measure[6], Remote desktop[7], IDOS[8], World Newspapers[9] a Network Scanner[10]. Tabulka 6.1 zobrazuje konkrétní data, která byla získána pomocí aplikace AppCheck. Jednotlivé analýzy byly prováděny na stejném zařízení ve stejné Wi-Fi síti. Každý záznam byl vytvořen po přibližně minutové aktivní práci s konkrétní aplikací.

Poté jsem uskutečnila několik na sobě nezávislých analýz, kde jedna z nich monitorovala aplikaci Spotify. V jejím případě se mi podařilo zaznamenat potencionálně nebezpečný paket. Určit jak moc závažná informace se v paketu nachází je automatizovaně poměrně náročné, což bylo již zmiňováno v kapitole 4.5. V tomto případě se jednalo o zaslání klíče aplikace — `appkey=21371601` — a id zařízení — `WZ7bhdelo%2FwDAGIZgohNJS8V` — pomocí protokolu HTTP za využití metody POST. Podle úvodní části URL, tj. `close`, se dá soudit, že aplikace tato data posílala při ukončení přenosu. Toto konkrétní zjištění bylo uskutečněno analýzou zaznamenanou v souboru `2019_04_25_08_47_46.pcap`, kde označení souboru určuje časové razítko záznamu ve formátu YYYY-MM-DD-HH-MM-SS.

[3]Viz https://play.google.com/store/apps/details?id=com.ninegag.android.app
[4]Viz https://play.google.com/store/apps/details?id=com.android.chrome
[5]Viz https://play.google.com/store/apps/details?id=com.Slack
[6]Viz https://play.google.com/store/apps/details?id=ml.kari.armeasure
[7]Viz https://play.google.com/store/apps/details?id=com.microsoft.rdc.android
[8]Viz https://play.google.com/store/apps/details?id=cz.mafra.jizdnirady
[9]Viz https://play.google.com/store/apps/details?id=com.KayaApps.newspapers
[10]Viz https://play.google.com/store/apps/details?id=com.easymobile.lan.scanner



Obrázek 6.2: První snímek zleva zobrazuje graf míry zabezpečení komunikace aplikace Spotify. Druhý obsahuje informace ke konkrétnímu paketu poslanému pomocí protokolu HTTP metodou POST. V URL se nachází odeslaný klíč aplikace a id zařízení. Zbylé dva obrázky patří k sobě a demonstrují, jak přibližně vypadá běžně zaznamenaná analytika. První z nich zobrazuje monitorované aplikace včetně počtu paketů, které odeslaly a přijaly. Druhý zobrazuje graf komunikace.

	Z	P	E
Analýza			
Pcap soubor zaznamenaného provozu	A	A	A
Míra bezpečnosti aplikací	A	A	A
Výpis monitorovaných aplikací	A	A	A
Rozlišení jednotlivých protokolů	N	A	A
HTTP dotazy	N	A	A
Velikost paketu	N	N	A
Časové razítko paketu	N	N	A
Obsah HTTPS paketu	N	N	A
Aplikace			
Ikonka	A	A	A
Název	A	A	A
Verze	A	A	A
Datum poslední aktualizace	N	A	A
Název balíku	N	A	A
UID	N	A	A
Udělená oprávnění	N	N	A
Síťová připojení			
Komunikující aplikace	A	A	A
Zdrojová IP adresa	A	A	A
Cílová IP adresa	A	A	A
Verze transportního protokolu	A	A	A
IP verze	A	A	A
Zdrojový port	N	A	A
Cílový port	N	A	A
Stav připojení	N	A	A
UID aplikace	N	A	A

Tabulka 6.2: Tabulka srovnání jednotlivých módů aplikace AppCheck vzhledem k jejich funkcionalitám. Názvy porovnávaných módů jsou zkráceny: Z = začátečník, P = pokročilý, E = expert. Zkratky hodnot tabulky: A =ANO, N = NE.

Kapitola 7

Závěr

Cílem této diplomové práce bylo vytvořit aplikaci pro bezpečnostní monitorování komunikace ostatních aplikací. Jednotlivé iterace vývoje byly automatizovaně testovány pomocí *unit* testů. Mimo to probíhalo i testování reálnými uživateli. Na základě jejich podnětů a připomínek byla aplikace vylepšována. Výsledná implementace umožňuje zaznamenat síťový tok konkrétní aplikace či více aplikací a poskytnout analýzu míry zabezpečení zaznamenaného toku. Mimo to poskytuje i informace o nainstalovaných aplikacích a ustanovených síťových připojeních, která jsou automaticky aktualizována každé $2\,s$.

Finální aplikaci jsem prezentovala na studentské konferenci Excel@FIT, kde jsem účastníky konference seznamovala s její funkcionalitou u svého plakátu. Reakce na aplikaci byly velmi pozitivní a návštěvníci z řad akademiků i odborné veřejnosti byli zaujati zejména tím, že kompletní funkcionalitu umí aplikace poskytnout bez nutnosti oprávnění uživatele *root*. I přes kladné ohlasy se však téměř všichni do jednoho shodli, že je aplikace více pro nadšence než pro širokou veřejnost. Pravděpodobně tento fakt spolu s enormním množstvím různých aplikací, které lze na Google Play stáhnout, přispívá tomu, že má aplikace velmi malou stahovanost.

Celkově se podařilo odhalit několik problémů s aplikacemi, které určitou část svých síťových požadavků řeší pomocí nezabezpečeného protokolu HTTP. Jednalo se například o známou a hodně rozšířenou aplikaci Spotify, která odesílala klíč aplikace a id zařízení. Díky tomu se dá konstatovat, že celkovým výstupem mé diplomové práce je plně funkční a užitečná aplikace, která umožňuje odhalit reálná bezpečnostní rizika mobilních aplikací a může tak přispět ke zvýšení bezpečnosti.

Literatura

[1] ABRAMSON, N.: THE ALOHA SYSTEM — Another alternative for computer communications. *Fall Joint Computer Conference*, November 1970: s. 281–285.

[2] Alqaydi, L.; Yeun, C. Y.; Damiani, E.: Security enhancements to TLS for improved national control. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec 2017, s. 274–279.

[3] Erickson, J.: *Hacking*. No Starch Press, vyd. 1. vydání, 2003, ISBN 1-59327-007-0.

[4] Fluhrer, S.; Mantin, I.; Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In *Selected Areas in Cryptography*, Springer, Berlin, Heidelberg, 2001, s. 1–24.

[5] H. Meyer, C.; M. Matyas, S.: *Cryptography: a new dimension in computer data security: a guide for the desing and implementation of secure systems*. John Wiley & Sons, 10 1982, ISBN 978-0471048923, 756 s.

[6] Huang, J.; Seberry, J.; Susilo, W.; aj.: Security Analysis of Michael: The IEEE 802.11i Message Integrity Code. In *Embedded and Ubiquitous Computing – EUC 2005 Workshops*, ročník 3823, 12 2005, s. 423–432.

[7] Hubbard, J.; Weimer, K.; Chen, Y.: A study of SSL Proxy attacks on Android and iOS mobile applications. 01 2014, ISBN 978-1-4799-2355-7, s. 86–91.

[8] Hái, J. F. Z.: Android capture tool development. May 2018, https://www.jianshu.com/p/ae4d433597ce.

[9] Jemerov, D.; Isakova, S.: *Kotlin in Action*. Manning Publications, první vydání, 2017, ISBN 9781617293290.

[10] Koopman, P.: 32-bit cyclic redundancy codes for Internet applications. In *Proceedings International Conference on Dependable Systems and Networks*, June 2002, s. 459–468.

[11] Lashkari, A. H.; Danesh, M. M. S.; Samadi, B.: A survey on wireless security protocols (WEP, WPA and WPA2/802.11i). In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, Aug 2009, s. 48–52.

[12] Li, G., Qing; Clark: *Security Intelligence: A Practitioner's Guide to Solving Enterprise Security Challenges*. John Wiley & Sons, Incorporated, 2015, ISBN 9781118896693.

[13] Misra, A.; Dubey, A.: *Android Security*. Auerbach Publications, první vydání, 2016, ISBN 9781439896464.

[14] Moskowitz, R.: Weakness in Passphrase Choice in WPA Interface. 11 2003.

[15] Pužmanová, R.: *Bezpečnost bezdrátové komunikace: jak zabezpečit wi-fi, bluetooth, GPRS či 3G*. Computer Press, vyd. 1. vydání, 2005, ISBN 80-251-0791-4.

[16] Sun, T.: Android VPN Service Explained with Packet Bypass Example Program. blogpost (english), Jun 2014.
URL https://www.thegeekstuff.com/2014/06/android-vpn-service/

[17] T. Fielding, R.: *Architectural Styles and the Design of Network - based Software Architectures*. Dizertační práce, UNIVERSITY OF CALIFORNIA, IRVINE, 01 2000.

[18] Tews, E.; Weinmann, R.-P.; Pyshkin, A.: Breaking 104 Bit WEP in Less Than 60 Seconds. In *Information Security Applications*, Lecture Notes in Computer Science, Springer, 2007, ISBN 978-3-540-77534-8, s. 188–202.

[19] Wang, X.; Yu, H.: How to Break MD5 and Other Hash Functions. *Lecture Notes in Computer Science*, ročník 3494, 05 2005: s. 561–561.

[20] Xu, Z.; Bai, K.; Zhu, S.: TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, ACM, 2012, ISBN 978-1-4503-1265-3, s. 113–124.

[21] Yan, K.: *Network protocols handbook*. Javvin, vyd. 2. vydání, 2005, ISBN 0-9740945-2-8.

Přílohy

Příloha A

Slovníček pojmů pro bezpečnostní protokoly

Bloková šifra – Bloková šifra je typ symetrické šifry, jenž dělí vstupní tok na daný počet bloků s pevně určenou délkou. Typicky je délka každého bloku 64 nebo 128 bitů. Není-li celková velikost vstupního textu dělitelná zvolenou délkou bloku, pak je nutné, aby se délka posledního bloku nadstavila. Využívá se k tomu tzv. výplň (anglicky *padding*), jenž dorovnává počet bitů do požadované délky bloku. Touto úpravou vznikne konečný počet stejně velkých bloků. Ty jsou následně kódovány symetrickým klíčem. Vstupní texty mohou být poměrně rozsáhlé, proto zde vzniká riziko znovupoužití klíče. Kvůli tomuto problému existuje několik různých režimů operací blokové šifry.

CRC – Zkratka CRC má v angličtině dva možné významy: *Cyclic redundancy checks* nebo *Cyclic redundancy code*. CRC je chyby detekující kód, který se používá pro zajišťování integrity dat. Může být implementovaný v hardware či software. Pro zjištění, zda se data přenosem nepoškodily, slouží porovnání rámcových kontrolních sekvencí (anglická zkratka FCS čili *Frame Check Sequence*). Vzájemně se porovnávají sekvence vypočítané před přenosem a po přenosu. Ty, které jsou vypočítané před přenosem (tj. na straně odesílatele), se přenáší společně s významovými daty. Sekvence po přenosu (tj. na straně příjemce) jsou počítané z příchozích dat stejným způsobem. Přenos považujeme za chybový, pokud se sekvence liší.

Man in the middle attack – Cílem útoku je obejít šifrovací mechanismy a stát se právoplatnou součástí komunikačního řetězce. Útočník se při útoku *man in the middle* snaží, aby se obě komunikující strany domnívaly, že stále posílají zprávy pouze správnému protějšku. Ve výsledku spravuje útočník dvě vzájemně oddělené šifrované komunikace. Zprávy si pro své potřeby uchovává a přeposílá je s co nejmenší latencí cílové komunikující straně. Tímto chováním se snaží předejít svému prozrazení.

Proudová šifra – Proudová šifra je jedním ze základních typů symetrických šifer. Zašifrování vstupního toku je řešeno pomocí kombinace s pseudonáhodným proudem bitů, tj. šifrovacím proudem (anglicky *cipher stream*), který se vytváří na základě šifrovacího klíče a konkrétního algoritmu. Proudová se jmenuje proto, že zašifrování probíhá postupně, podle pořadí načítání jednotlivých znaků. U většiny proudových šifer platí, že jsou poměrně snadno rozluštitelné. Nicméně samotný princip proudové šifry je dostatečně bezpečný. Jejich hlavní slabinou je inicializační vektor. Ten představuje počáteční

nastavení registrů, které se používá pro šifrování i dešifrování přenášených zpráv. Problém nastává, je-li vektor použitý vícekrát, což je typické u krátkých inicializačních vektorů.

Replay attack – Útok pomocí opakování. Využívá se při něm slabiny bezpečnostního protokolu, spočívající v nedostatečné či žádné kontrole správné návaznosti paketů. Útočník odchytí správný paket, následně do něj může uložit škodlivé informace a odeslat příjemci. Paket se pak jeví jako zpožděný. Bez správného ošetření těchto paketů ho příjemce bez problému přijme.

Příloha B

Verze Linux jádra dle verzí systému Android

Následující tabulka uvádí rozdělení jednotlivých verzí Android jádra v závislosti na tom, ze které verze Linux kernelu čerpá. Zkratka AOSP zde vyjadřuje *Android Open Source Project*. Tento výpis čerpá hlavně z oficiální dokumentace Androidu[1].

Verze Androidu	**API level**	**Linux kernel v AOSP**
1.5 Cupcake	3	2.6.27
1.6 Donut	4	2.6.29
2.0/1 Eclair	5-7	2.6.29
2.2.x Froyo	8	2.6.32
2.3.x Gingerbread	9, 10	2.6.35
3.x.x Honeycomb	11-13	2.6.36
4.0.x Ice Cream San	14, 15	3.0.1
4.1.x Jelly Bean	16	3.0.31
4.2.x Jelly Bean	17	3.4.0
4.3 Jelly Bean	18	3.4.39
4.4 Kit Kat	19, 20	3.10
5.x Lollipop	21, 22	3.16.1
6.0 Marshmallow	23	3.18.10
7.0 Nougat	24	4.4.1
7.1 Nougat	25	4.4.1
8.0 Oreo	26	4.10
8.1 Oreo	27	4.10
9.0 Pie	28	4.4, 4.9 and 4.15

Tabulka B.1: Tabulka srovnání verze Android jádra vůči použitému výchozímu Linux jádru. Tabulka srovnání verzí jádra Androidu vůči jádru Linuxu, ze kterého vychází.

[1] Stránky dokumentace: https://android.googlesource.com/platform/external/kernel-headers/