VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÍ SYSTÉMU PRO SHLUKOVOU ANALÝZU BINÁRNÍCH SOUBORŮ

IMPROVING AN EXISTING SYSTEM FOR CLUSTERING BINARY FILES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PAVOL PLASKOŇ**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Dr. Ing. DUŠAN KOLÁŘ**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/20059/2016/xplask00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Plaskoň Pavol**
Obor: Informační technologie
Téma: **Rozšíření systému pro shlukovou analýzu binárních souborů**
Improving an Existing System for Clustering Binary Files
Kategorie: Překladače

Pokyny:

1. Studujte metody shlukové analýzy dat a jejich možné využití při analýze binárních souborů.
2. Seznamte se se systémem Clusty pro shlukovou analýzu souborů společnosti AVG Technologies, který se používá pro účely analýzy potenciálně škodlivých souborů.
3. Navrhněte takové analýzy a postupy, které rozšíří stávající systém o nové možnosti, např. podpora dalších souborových formátů vstupních dat, shlukování na základě dalších rysů souborů atd.
4. Po konzultaci s vedoucím a konzultantem implementujte analýzy navržené v předchozím bodě.
5. Výsledky práce důkladně otestujte z pohledu přesnosti a rychlosti analýzy nad reálnými vzorky.
6. Zhodnoťte svou práci a diskutujte možný budoucí vývoj.

Literatura:

- Zendulka, J. a kol.: Získávání znalostí z databází. FIT VUT v Brně, 160 s., 2009. (elektronicky)
- Everitt, B.: Cluster Analysis. Hoboken: Wiley, 2011. ISBN 978-0-470-97844-3
- Interní dokumentace systému Clusty, AVG Technologies, 2016.
- Dle doporučení vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- První tři body zadání a alespoň rozpracovaný čtvrtý bod.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kolář Dušan, doc. Dr. Ing.**, UIFS FIT VUT
Konzultant: Zemek Petr, Ing., AVG
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

S rastúcim množstvom informácií sa zvyšuje potreba sumarizovať dáta. Jednou z používaných metód je aj zhluková analýza. Je to proces triedenia objektov do skupín na základe spoločných vlastností. V antivírusových firmách sa denne analyzuje veľké množstvo súborov. S cieľom urýchliť túto analýzu vznikol aj v AVG Technologies nástroj na zhlukovú analýzu binárnych súborov. Táto práca popisuje návrh a implementáciu rozšírenia existujúceho nástroja o zhlukovú analýzu APK súborov a s ním súvisiacich DEX súborov. Cieľom je analyzovať dané súborové formáty, vybrať z nimi poskytovaných informácií vhodné charakteristiky a vytvoriť heuristiky pre ich zhlukovú analýzu. Okrem podpory pre APK a DEX súbory bol nástroj rozšírený aj o zhlukovú analýzu archívov. Všetko bolo otestované a nasadené do produkčnej verzie.

Abstract

The increase in the amount of information requires advanced processing of data. One of such methods is cluster analysis. It is a process of classification of objects based on their similarity. Anti-malware companies analyze large amount of files every day. In order to speed up their analysis, a cluster-analysis tool was implemented in AVG Technologies. The goal of this work is to improve this tool for clustering binary files by adding support and heuristics for cluster analysis of APK and DEX file formats. Apart from the newly added support for APK and DEX files, the tool has been extended to support cluster analysis of archives. Everything was tested and put into production.

Kľúčové slová

zhluková analýza, statická analýza, dynamická analýza, Android, APK, DEX, ZIP

Keywords

cluster analysis, static analysis, dynamic analysis, Android, APK, DEX, ZIP

Citácia

PLASKOŇ, Pavol. *Rozšíření systému pro shlukovou analýzu binárních souborů*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kolář Dušan.

Rozšíření systému pro shlukovou analýzu binárních souborů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Doc. Dr. Ing. Dušana Koláře. Ďalšie informácie mi poskytli Petr Zemek a Jakub Křoustek. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Pavol Plaskoň
11. mája 2017

Poďakovanie

Rád by som poďakoval svojmu vedúcemu Doc. Dr. Ing. Dušanovi Kolářovi a hlavne Ing. Petrovi Zemkovi a Ing. Jakubovi Křoustkovi za konzultácie, spätnú väzbu a odborné rady.

Obsah

Kapitola 1

Úvod

Klasifikácia, teda rozdeľovanie objektov do skupín podľa rôznych kritérií, je základom mnohých vedeckých odvetví. V biológií už Aristoteles rozčlenil zvieratá na tie, ktoré majú krv, a tie bez krvi (teda aspoň bez červenej krvi). Jeho rozdelenie je veľmi blízke dnešnému deleniu na stavovce a bezstavovce. Existuje aj samostatný odbor pre klasifikáciu živých organizmov, nazývaný *taxonómia* [9]. S cieľom poskytnúť objektívnu a stabilnú klasifikáciu vznikli rôzne numerické metódy. Dnes sa obecne metódam pre hľadanie skupín zhlukov v dátach hovorí *zhluková analýza* (angl. *cluster analysis*). Okrem biológie našla zhluková analýza uplatnenie aj v psychológii, sociálnych vedách, astronómii, strojovom učení atď.

Zhluková analýza vytvára skupiny objektov na základe informácií získaných len z daných dát [27]. Objekty v rámci skupiny by mali mať k sebe čo najbližšie, a ďaleko od objektov iných skupín. Vo väčšine metód sa hľadá rozdelenie dát, v ktorom patrí každý objekt práve do jedného zhluku. V niektorých prípadoch môžu vznikať aj prekrývajúce sa zhluky s lepšími výsledkami. U *fuzzy* zhlukovania dokonca jeden objekt patrí do všetkých zhlukov. Podrobnejšie je zhluková analýza popísaná v kapitole 2. Táto bakalárska práca sa zaoberá zhlukovou analýzou binárnych súborov pre využitie pri analýze škodlivého softvéru (angl. *malware*).

Každým rokom sa zvyšuje počet vírusov, čo prináša stále vyššie nároky na ochranu počítačov, mobilných telefónov a ďalšej elektroniky. Firmy poskytujúce ochranu pred vírusmi musia s nimi držať krok. Denne dostávajú státisíce vzoriek na analýzu [13]. To si samozrejme vyžaduje automatizáciu. Avšak automatická detekcia má niekoľko prekážok, ako obfuskovanie, kompresia alebo šifrovanie [11]. Antivírusové firmy využívajú nástroje na automatickú analýzu, zhlukovanie ale aj klasifikáciu vzoriek. Medzi tieto nástroje patrí aj *Clusty* – nástroj vyvíjaný v AVG Technologies (teraz to už spadá pod firmu Avast).

Hlavným účelom nástroja *Clusty* je prvotná analýza prichádzajúcich vzoriek, zisťovanie podobností medzi nimi a následné rozdelenie do skupín na základe vhodných heuristík. Analytik takto dostane celé skupiny podobných vzoriek. Analýzou jednej vzorky môže priradiť celú skupinu k danej rodine vírusov (u vírusov je zaužívaný termín *rodina* namiesto triedy). Nemusí teda opakovane skúmať množstvo mierne odlišných vzoriek, ktoré autori len čiastočne upravili v snahe vyhnúť sa detekcii antivírusovými programami. Podrobnejší popis nástroja je popísaný v kapitole 3.

Cieľom tejto práce je rozšíriť existujúci nástroj pre zhlukovú analýzu binárnych dát o podporu ďalších formátov. Najväčšie zastúpenie medzi denne prichádzajúcimi vírusmi má formát PE. Je to pochopiteľné vzhľadom na rozšírenosť operačného systému Windows. S príchodom inteligentných telefónov sa otvoril nový priestor pre tvorbu vírusov. S ich rastúcou popularitou stúpa aj záujem útočníkov. Ide hlavne o nárast škodlivých súborov vo

formáte APK využívanom na platforme Android. S týmto súborovým formátom súvisí aj archivovací formát ZIP a formát DEX, ako je podrobnejšie popísané v kapitole 4. Populárne sú aj ďalšie archívy (TAR, RAR, 7-ZIP, ...), ktoré zatiaľ nemajú v nástroji Clusty podporu. V kapitole 5 je popísaný návrh rozšírení a ich zapojenie do nástroja. Samotná implementácia je ďalej uvedená v kapitole 6. V tejto práci ide o nájdenie vhodných charakteristík v APK súboroch pre zhlukovú analýzu. Nejde o vytvorenie nástroja na analýzu APK formátu.

Počas práce je nutné neustále testovanie. Či už ide o jednotkové testy implementácie, pamäťovú a časovú náročnosť, ale aj celkové výsledky zhlukovania na overenie úspešnosti heuristík. Na to sa využijú vzorky už rozdelené do rodín a porovnajú sa dosiahnuté výsledky. Testovanie je popísané v kapitole 7, výsledky sú zhodnotené v kapitole 8 a celá práca, aj s navrhnutými rozšíreniami, je zhrnutá v záverečnej kapitole 9.

Kapitola 2

Zhluková analýza

Táto kapitola vychádza z [9, 22, 31].

S neustále rastúcim množstvom a veľkosťou databáz v rôznych odvetviach sa zvyšuje potreba sumarizovať dáta. Proces analyzovania dát rôznymi technikami z rôznych perspektív sa už často označuje ako dolovanie informácií z dát (angl. *data mining*), podrobnejšie popísaný v [30]. Zhlukovanie je proces rozdeľovania objektov do tried (zhlukov, angl. *clusters*) za základe podobnosti. Objekty v rámci triedy sú si veľmi podobné a zároveň nie sú príliš podobné objektom v iných triedach. Podobnosť sa posudzuje na základe hodnôt vybraných atribútov a často sa využívajú vzdialenostné funkcie. Z hľadiska strojového učenia ide o učenie bez učiteľa. Nevyžaduje teda preddefinované triedy ani trénovanie na množine objektov už rozdelených do tried [31].

Vybrané vlastnosti zhlukovacích metód:

- škálovateľnosť – algoritmy často dosahujú dobré výsledky na malom objeme dát, v praxi je ale neraz potrebné analyzovať veľké množstvo dát.
- spracovanie rôznych typov atribútov – základným a najjednoduchším typom na spracovanie sú numerické dáta. Často je potrebné spracovať aj iné typy, ako binárne, textové, . . .
- tvorba zhlukov rôzneho tvaru – najbežnejšie metódy využívajú Euklidovskú alebo Manhattanovksú vzdialenosť, čím vznikajú zhluky kruhovitého tvaru. Iný tvar by mohol lepšie odpovedať hľadaným triedam.
- požiadavky na znalosť problému pri určovaní parametrov – niektoré metódy požadujú vstupné parametre (napr. požadovaný počet zhlukov). To môže výrazne ovplyvniť kvalitu výsledkov, pretože nie vždy sa dá dopredu odhadnúť výsledný počet zhlukov.
- schopnosť vyrovnať sa so šumom – objekty s chybnými alebo neznámymi dátami môžu znížiť kvalitu výsledkov.
- citlivosť na poradie vstupných objektov – niektoré algoritmy vytvoria iné zhluky nad rovnakou sadou objektov pri ich rôznom usporiadaní.
- schopnosť spracovávať vysoko-dimenzionálne dáta – bežné metódy pracujú dobre nad dátami s dvoma, troma atribútmi.
- zhlukovanie s obmedzeniami – cieľom je nájsť triedy, ktoré spĺňajú zadané obmedzenia.
- použiteľné zhluky – výsledky musia byť interpretovateľné, zrozumiteľné a použiteľné.

2.1 Typy dát

Najčastejšie používané dátové štruktúry v zhlukovacích metódach sú:

- dátová matica – viacrozmerná dátová matica reprezentujúca n objektov a ich p atribútov. Reprezentovaná relačnou tabuľkou alebo maticou $n \times p$. Záznam o_{ij} reprezentuje hodnotu j-tého atribútu v i-tom objekte.

$$\begin{bmatrix} o_{11} & o_{12} & \dots & o_{1p} \\ o_{21} & o_{22} & \dots & o_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ o_{n1} & o_{n2} & \dots & o_{np} \end{bmatrix}$$

- matica podobnosti – obsahuje vzdialenosti pre všetky dvojice objektov. Je reprezentovaná tabuľkou $n \times n$. Záznam $d(i, j)$ udáva vzdialenosť i-tého a j-tého objektu. Pre zhodné objekty je vzdialenosť nulová. Hodnota na diagonále je vzdialenosť prvku od seba samého, teda $d(i, i) = 0$. Matica je trojuholníková, pretože $d(i, j) = d(j, i)$.

$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ d(n,1) & d(n,2) & \dots & \dots & 0 \end{bmatrix}$$

Objektmi zhlukovania môžu byť ľudia, tovar, mestá, choroby, atď. Každý objekt je popísaný p atribútmi. Otázkou je, ako sa určí vzdialenosť medzi objektmi, keďže atribúty môžu byť rôznych typov. Typicky sú to nasledujúce typy premenných:

- binárne – hodnoty 0/1, áno/nie. Sú dva typy týchto premenných:
  - symetrické – oba stavy majú rovnakú pravdepodobnosť (napríklad pohlavie). Vzdialenosť môže byť vyjadrená koeficientom zhody:

$$d(i, j) = \frac{r + s}{q + r + s + t}$$

    kde r je počet objektov s hodnotou 1 pre objekt i a hodnotou 0 pre objekt j, s je počet premenných s hodnotou 0 pre objekt i a hodnotou 1 pre j, q je počet premenných s hodnotou 1 pre oba objekty a t je počet premenných s hodnotou 0 pre oba objekty.
  - asymetrické – stavy nemajú rovnakú pravdepodobnosť. Nech má stav 1 väčšiu váhu, potom je zhoda dvoch hodnôt 1 významnejšia než zhoda dvoch hodnôt 0. Vzdialenosť sa môže určiť Jaccardovým koeficientom, kde parameter t sa nevyskytuje kvôli nízkej dôležitosti zhody dvoch 0:

$$d(i, j) = \frac{r + s}{q + r + s}$$

- nominálne – sú to zobecnené binárne premenné, ale hodnoty majú obmedzený počet (napríklad farba očí). Vzdialenosť môže byť vyjadrená koeficientom zhody:

$$d(i, j) = \frac{n - m}{n}$$

kde m je počet premenných s rovnakou hodnotou pre oba objekty a n je počet všetkých premenných. Premenným navyše môžu byť priradené váhy.

- ordinálne – podobné nominálnym, ale hodnoty sú usporiadané. Napríklad výška: nízky, stredný, vysoký. Vzdialenosť môže byť určená niektorou zo vzdialenostných funkcií, ale najprv je potrebné previesť hodnotu f všetkých premenných na hodnotu $z \in \langle 0, 1 \rangle$, aby mali rovnakú váhu:

$$z_{if} = \frac{r_{if} - 1}{M_f - 1}$$

kde M_f je počet hodnôt, ktoré môže premenná f nadobúdať a $r_{if} \in \{1, \ldots, M_f\}$ je číselná hodnota zastupujúca hodnotu premennej f.

- intervalové – napríklad teplota (Celsius), čas. Pre výpočet vzdialenosti sa najčastejšie používajú tieto funkcie:
  - Euklidovská:

$$d(i, j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + \cdots + (x_{in} - x_{jn})^2}$$

    kde $i = (x_{i1}, x_{i2}, \ldots, x_{in})$ a $j = (x_{j1}, x_{j2}, \ldots, x_{jn})$ sú n-dimenzionálne objekty.
  - Manhattanovská:

$$d(i, j) = |x_{i1} - x_{j1}| + |x_{i2} - x_{j2}| + \cdots + |x_{in} - x_{jn}|$$

  - Minkowského:

$$d(i, j) = (|x_{i1} - x_{j1}|^p + |x_{i2} - x_{j2}|^p + \cdots + |x_{in} - x_{jn}|^p)^{\frac{1}{p}}$$

    kde p je koeficient typicky s hodnotu 1 alebo 2 (Euklidovská vzdialenosť).

- pomerové – podobné intervalovým premenným, ale navyše hodnota 0 vyjadruje neexistenciu atribútu (napríklad výška, váha). Vzdialenosť je možné určiť viacerými spôsobmi:
  - rovnako ako pre intervalové premenné (môže dôjsť ku skresleniu)
  - logaritmickou transformáciou $y_{if} = \log x_{if}$ a výsledné hodnoty spracovať ako intervalové premenné
  - premenné spracovať ako ordinálne hodnoty a ich rozsah ako intervalové

2.2 Výpočet podobnosti

Okrem vzdialenostných funkcií uvedených pri jednotlivých typoch dát v sekcii 2.1 sa vyžívajú aj funkcie podobnosti. Následujúce funkcie sú prevzaté z [22]. Budú definované nad dvoma vektormi – x_i a x_j. $\bar{x}$ vyjadruje aritmetický priemer hodnôt vektora x a x^T je transpozícia vektora x.

Kosínusová podobnosť

Táto funkcia je vhodná v prípade, ak uhol medzi dvoma vektormi rozumne vyjadruje ich podobnosť:

$$s(x_i, x_j) = \frac{x_i^T \cdot x_j}{\|x_i\| \cdot \|x_j\|}$$

Pre množiny A a B je definovaná kosínusová podobnosť takto:

$$s(A, B) = \frac{|A \cap B|}{\sqrt{|A| \cdot |B|}}$$

Pearsonova podobnosť

Normalizovaná Pearsonova podobnosť je definovaná takto:

$$s(x_i, x_j) = \frac{(x_i - \bar{x}_i)^T \cdot (x_j - \bar{x}_j)}{\|x_i - \bar{x}_i\| \cdot \|x_j - \bar{x}_j\|}$$

Rozšírená Jaccard funkcia podobnosti

$$s(x_i, x_j) = \frac{x_i^T \cdot x_j}{\|x_i\|^2 + \|x_j\|^2 - x_i^T \cdot x_j}$$

Pre množiny A a B je Jaccard funkcia definovaná takto:

$$s(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Pre výslednú podobnosť platí: $0 \leq s(A, B) \leq 1$. Pre prázdne množiny A a B je podobnosť $s(A, B) = 1$.

Dice koeficient podobnosti

$$s(x_i, x_j) = \frac{2x_i^T \cdot x_j}{\|x_i\|^2 + \|x_j\|^2}$$

Pre množiny A a B je funkcia definovaná takto:

$$s(A, B) = \frac{2|A \cap B|}{|A| + |B|}$$

Táto funkcia je podobná Jaccard funkcii, ale nespĺňa trojuholníkovú nerovnosť, čo je jeden z axiómov vzdialenostných funkcií. Jednoduchým dôkazom sú množiny $\{a\}$, $\{b\}$ a $\{a, b\}$ (trojuholníková nerovnosť platí pre vzdialenosť, v tomto prípade je to $d = 1 - s(A, B)$).

2.3 Typy metód

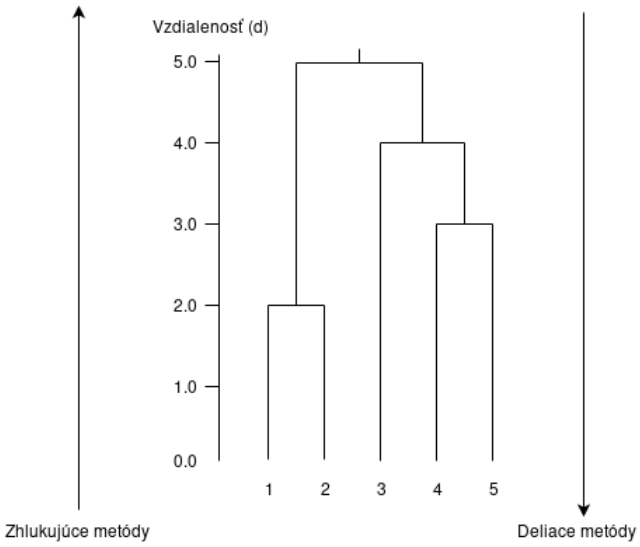
Výber konkrétnej metódy závisí na type spracovávaných dát a účele analýzy. Nedá sa obecne určiť najvhodnejšia metóda.

Hierarchické metódy

Výsledkom týchto metód je hierarchický rozklad množiny objektov (strom zhlukov). Reprezentuje sa často dendogramom, ukážka je na obrázku 2.1. Podľa priebehu rozkladu sa delia na:

- zhlukujúce metódy – každý objekt je reprezentovaný vlastnou triedou a následne sa najpodobnejšie triedy zlučujú, pokým sa nespoja do jedného zhluku alebo nie je dosiahnutá požadovaná úroveň. Tento prístup využíva väčšina hierarchických metód [31].
- deliace metódy – všetky objekty sú súčasťou jedného zhluku. Ten sa následne rozdeľuje na menšie, pokým nie je každý objekt v samostatnom zhluku alebo nie je dosiahnutá požadovaná štruktúra.

Nevýhodou týchto metód je kvadratická zložitosť a nemožnosť vrátiť sa o krok späť. Čo bolo raz rozdelené, sa už nedá spojiť (u zhlukujúcich metód naopak, spojené objekty sa už nerozdelia).



Obr. 2.1: Dendogram výsledku hierarchických metód (prevzatý z [9])

Deliace metódy

Tieto metódy rozdeľujú n objektov do k tried ($k \leq n$). Každá trieda musí obsahovať aspoň jeden objekt a objekt patrí práve do jednej triedy. Pre tieto metódy je nutné zadať počet tried, do ktorých chceme objekty rozdeliť. Nutnosť špecifikovať počet tried je ich nevýhodou [31].

Postup delenia je následovný. Najprv sa vyberie k objektov, ktoré reprezentujú počiatočné triedy a zvyšné objekty sa na základe vzdialenosti rozdelia do týchto tried. Potom sa v každej triede vyberie reprezentant a objekty sa presúvajú medzi triedami tak, aby vzdialenosť objektov z jednej triedy bola maximálna a z rôznych tried minimálna. Najpoužívanejšie heuristiky sú *k-means* a *k-medoids*. Výsledné zhluky majú guľovitý tvar [31].

K-means je metóda založená na centrálnom bode. Objekt reprezentujúci triedu je väčšinou fiktívny objekt, ktorého atribúty sú strednou hodnotou atribútov všetkých objektov

danej triedy. Metóda nenájde zhluky nekonvexného tvaru, je citlivá na šum a odľahlé objekty. Veľkou výhodou oproti iným metódam je lineárna zložitosť [22].

K-medoids je metóda založená na reprezentujúcom objekte. Na rozdiel od k-means metódy je objekt reprezentujúci triedu vždy skutočný objekt danej triedy, najbližší k jej stredu. V porovnaní s k-means je silnejšia, pretože znižuje vplyv odľahlých objektov a šumu, ale je výpočtovo náročnejšia.

Metódy založené na mriežke

Tieto metódy využívajú viacúrovňovú mriežkovú dátovú štruktúru. Priestor je rozdelený mriežkou na bunky a všetky operácie prebiehajú nad touto mriežkou. Každá bunka obsahuje informácie o skupine objektov v nej.

Algoritmy typicky obsahujú tieto kroky [7]:

1. Vytvorenie mriežky – rozdelenie priestoru na bunky.
2. Výpočet hustoty v každej bunke.
3. Roztriedenie buniek podľa hustoty.
4. Identifikácia stredu pre každý zhluk.
5. Prechod susedných buniek.

Hlavnou výhodou týchto metód je nízka časová náročnosť. Platí to aj pre veľké množiny dát [31]. K týmto metódam patria napríklad *WaveCluster*, *CLIQUE* a *STING*. Kvôli výpočtu hustoty buniek sa môžu niektoré algoritmy zaradiť k algoritmom založeným na hustote.

Metódy založené na modeloch

Cieľom týchto metód je nájsť zhluky, ktoré budú odpovedať nejakému matematickému modelu. Najčastejšie ide o dáta generované podľa nejakej zloženej pravdepodobnostnej distribučnej funkcie. Každý zhluk bude reprezentovaný parametrizovanou pravdepodobnostnou distribučnou funkciou. Výsledný model je potom tvorený z k takýchto funkcií (pre každý zhluk jedna). Problémom je nájsť vhodné parametre týchto funkcií.

- *Expectation Minimization* – metóda podobná algoritmu *k-means*. Každému objektu sa určí pravdepodobnosť príslušnosti do zhluku, v ktorom sa nachádza. Tieto hodnoty sa potom použijú na výpočet parametrov nových bodov reprezentujúcich zhluky.
- konceptuálne zhlukovanie – metóda, ktorá sa snaží nájsť klasifikačné schéma pre objekty a charakteristický popis pre každý zhluk. Patrí sem napríklad metóda *COBWEB*.
- neurónové siete – každý zhluk je reprezentovaný neurónom (prototyp). Vstupné dáta sú taktiež reprezentované neurónmi, ktoré sú spojené s prototypom. Každé spojenie má váhu, ktorá sa upravuje počas učenia [22].

Metódy pre vysoko-dimenzionálne dáta

S rastúcim počtom dimenzií je len ich malé množstvo relevantné pre zhlukovanie. Ďalšie dimenzie môžu produkovať šum a znemožniť identifikáciu zhluku. Taktiež dochádza k väčšiemu rozptýleniu dát. Tieto problémy sa snaží riešiť napríklad metóda výberu atribútov.

Výberom atribútov sa odstránia nepodstatné dimenzie (atribúty). Ponechané dimenzie sú dostatočne relevantné pre vytvorenie zhlukov. Najčastejšie s využitím strojového učenia s učiteľom sa prechádzajú podmnožiny atribútov a hľadajú sa relevantné atribúty podľa ohodnotenia jednotlivých tried objektov. Ďalšou možnosťou je analýza podmnožín pomocou entropie.

Medzi tieto metódy patrí napríklad *CLIQUE*, *PROCLUS* [31].

Metódy založené na hustote

U týchto metód je zhluk oblasťou s veľkou hustotou objektov v priestore dát, ktorá je od iných takýchto oblastí oddelená oblasťami s malou hustotou dát (označované ako šum). Tieto metódy majú dobré výsledky aj na dátach s výskytom šumu a odľahlých objektov. Výsledné zhluky môžu mať rôzny tvar.

DBSCAN

Metóda DBSCAN (Density-Based Spatial Clustering of Applications with Noise) patrí k metódam založeným na hustote [9]. Vytvára zhluky z oblastí s dostatočne veľkou hustotou objektov.

Algoritmus vyžaduje dva parametre: ε a min_points, kde ε je maximálna vzdialenosť medzi dvomi objektmi, aby mohli byť v jednom zhluku, a min_points určuje minimálny počet objektov v zhluku. Hodnota ε určuje hustotu – čím nižšia vzdialenosť, tým vyššia hustota.

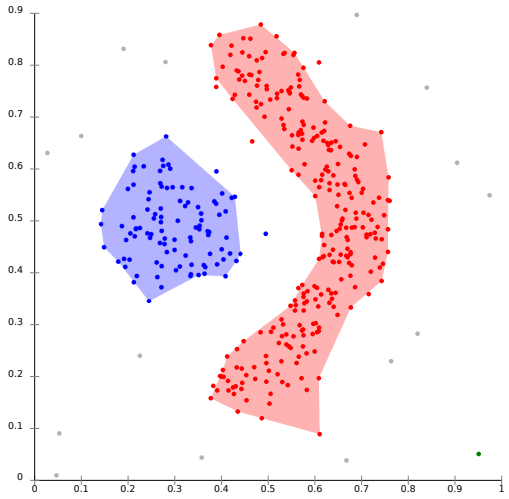
Výhodou je, že medzi parametrami algoritmu nie je počet zhlukov, ako napríklad v *k-means* algoritme [11]. Taktiež vytvára zhluky aj iných tvarov než kruhové (viď obr. 2.2). Dokáže nájsť aj nelineárne-separovateľné zhluky. Jednou z nevýhod môžu byť horšie výsledky pri väčších dimenziách [9] a taktiež je dôležité správne nastavenie hodnôt min_points a ε.

Algoritmus pracuje nasledujúcim spôsobom. Pre každý objekt zistí, koľko iných objektov sa nachádza v jeho ε-okolí (ich vzdialenosť je menšia alebo rovná ε). Ak je ich počet aspoň min_points, tak sa daný objekt označuje ako jadro a vytvorí sa okolo neho zhluk. Rekurzívne sa prejdú všetky jeho objekty, a pripojí sa ich ε okolie k danému zhluku, ak obsahuje aspoň min_points objektov. Objekty, ktoré nezapadnú do žiadneho zhluku, sa označujú za šum.

Soft-computing metódy

Medzi tieto metódy patria napríklad *fuzzy* metódy. U nich už neplatí, že objekt patrí práve do jedného zhluku, ale patrí do každého zhluku s určitou mierou príslušnosti.

Ďalšími algoritmami sú metódy inšpirované prírodou, genetické algoritmy inšpirované chromozómami, prirodzeným výberom, mutáciou, a ďalšie [22].



Obr. 2.2: Ukážka zhlukov vytvorených algoritmom DBSCAN (prevzatá z [28])

2.4 Zhluková analýza vírusov

Existujú dva základné princípy analýzy vírusov – statická a dynamická analýza. Obe metódy sú popísané v tejto sekcii.

V zhlukovej analýze je možné využiť oba prístupy a taktiež ich kombináciu. Oba prístupy majú svoje výhody aj nevýhody. U statickej analýzy nie je potrebné spúšťať program. Navyše ak ide o skriptovací jazyk, tak kód sa dá jednoducho prečítať, nie je potrebný preklad do assembleru alebo vyššieho jazyka. Problémom je ale obfuskovanie. To sa snaží riešiť dynamická analýza. V tomto prístupe sa vírus spustí vo virtuálnom stroji alebo emulovanom prostredí a monitoruje sa jeho správanie – volanie systémových funkcií, komunikácia so vzdialeným serverom a pod. [25].

Niekedy sa tieto dve techniky skombinujú do hybridnej analýzy. Zachovajú sa tak výhody oboch metód a zároveň sa zmiernia ich slabiny. Podrobnejšie sa tomuto prístupu venuje [23].

2.4.1 Statická analýza

Nasledujúci popis vychádza z [16].

Vo väčšine prípadov nie sú k dispozícii zdrojové kódy, len binárny súbor. Niektoré informácie zo zdrojového kódu sa pri preklade vypúšťajú a to komplikuje analýzu binárneho kódu. Prvým krokom statickej analýzy je zvyčajne preklad binárneho kódu do jazyka symbolických inštrukcií (angl. *assembly language*, *assembler*). Nástroj pre tzv. spätný preklad sa označuje ako *disassembler*. Ani tento prevod nie je triviálny a musí sa vysporiadať s množstvom prekážok. Medzi algoritmy spätného prevodu patria napríklad:

- lineárny prechod – prechádza vstupný súbor bajt po bajte. Inštrukcie ale nie sú vždy jedna za druhou. Pri lineárnom prechode sa teda môžu chybne dekódovať aj bajty použité len na zarovnanie alebo konštanty uložené v časti pre kód (GCC optimalizácia). Tento prístup sa využíva napríklad v GNU `objdump` [14].

- rekurzívny prechod – algoritmus, ktorý sa snaží dekódovať len významné bajty a identifikovať všetky možné skoky, čím skúma len programovo dostupné adresy. V prípade,

že sa nedá cieľ skoku určiť staticky (adresa je v registri), môže tento prístup ignorovať celé bloky kódu. Navyše je potrebné poznať sémantiku inštrukcií kvôli rozpoznaniu zmeny toku kódu.

- hybridný prístup – kombinuje výhody oboch predchádzajúcich metód. Najprv sa použije rekurzívny prechod a potom sa neanalyzované časti kódu prejdú lineárne. Taktiež sa môžu oba prístupy spojiť a iteratívne analyzovať celý kód [14].

Existuje mnoho komerčných aj voľne šíriteľných disassemblerov. Medzi najpopulárnejšie patrí IDA Pro[1], OllyDbg[2], objdump ...

Získaný kód v jazyku symbolických inštrukcií sa analyzuje rôznymi technikami:

- analýza riadenia toku – využíva graf riadenia toku, kde uzly reprezentujú základné bloky kódu a hrany sú prechody medzi blokmi. Vyžaduje si presné identifikovanie všetkých volaní skoku (volanie funkcií aj podmienené a nepodmienené skoky) a cieľov skoku.

- analýza toku dát – cieľom je získať množinu možných hodnôt v danom mieste programu. Zobrazuje sa grafom toku dát, ktorý vyjadruje závislosť hodnôt medzi jednotlivými úsekmi kódu.

- identifikácia vzorov – logicky súvisiaca skupina inštrukcií.

- program slicing – využíva graf riadenia toku aj graf toku dát. Na základe kritéria $C \equiv (s, V)$, kde s je uzol z grafu toku dát a V je podmnožina premenných, vytvorí množinu inštrukcií S_C (slice), ktoré sú relevantné pri výpočte hodnôt premenných z V. Analýza sa teda môže zamerať len na vybranú skupinu inštrukcií [6].

- entropia – vysoká hodnota entropie zvyčajne znamená použitie komprimácie a šifrovania.

Ďalšou možnosťou okrem prekladu do symbolických inštrukcií je preklad do vyššej úrovne, teda spätný preklad do zdrojového kódu (angl. *decompilation*). Tento proces zahŕňa aj prevod binárneho súboru do jazyka symbolických inštrukcií. Navyše sa musí vysporiadať s chýbajúcimi informáciami, ktoré prekladač zahodil. Z toho vyplýva, že spätný preklad je oveľa náročnejší a presná replika pôvodného zdrojového súboru sa nedá dosiahnuť. Na druhej strane, funkcionalita zrekonštruovaného kódu môže byť zhodná s originálom, a navyše, zdrojový kód (napríklad v jazyku C) môže byť čitateľnejší než strojový kód.

V AVG Technologies je vyvíjaný spätný prekladač RetDec[3].

2.4.2 Dynamická analýza

Dynamická analýza sleduje správanie programu za behu. Využíva sa pri hľadaní chýb v programe, profilovanie (odhalenie pomalých častí), odhalenie nebezpečného správania, atď.

Pri analýze vírusov sa kvôli riziku poškodenia systému často využívajú tieto techniky:

- sandboxing – bezpečnostný mechanizmus, ktorý izoluje beh procesu od zvyšku systému. Systému nehrozí žiadne riziko ani v prípade, kedy ide o spustenie vírusu.

[1] https://www.hex-rays.com/products/ida/
[2] http://www.ollydbg.de/
[3] https://retdec.com/

- emulácia – virtuálne prostredie, ktoré môže vytvoriť ľubovoľnú architektúru, nezávisle na hostiteľskom stroji. Inštrukcie emulovaného stroja sa prekladajú do inštrukcií hostiteľa. Nevýhodou teda môže byť vyššia zložitosť a nižšia rýchlosť kvôli tomuto prekladu.

- virtualizácia – zjednodušene povedané, podobá sa emulácii, ale inštrukčná sada virtualizovaného systému musí byť kompatibilná s hostiteľským strojom [14]. Inštrukcie nevyžadujúce privilegovaný mód teda môžu byť spustené na reálnom stroji a to prináša zrýchlenie do virtuálneho stroja. Privilegovaný mód reálneho stroja ale nie je prístupný a existuje iba virtuálne [8].

Pri dynamickej analýze vírusov sa sleduje podozrivé správanie programu, hlavne volanie systémových funkcií, aplikačného rozhrania (API), použité dynamické knižnice, vstupno-výstupné operácie, sieťová komunikácia atď. Využíva sa pri tom napríklad odchytávanie volaní funkcií. Cieľom je zaznamenať napríklad hodnoty parametrov funkcie a jej návratovú hodnotu.

Zdrojový kód vírusov nie je k dispozícii takmer nikdy. Pri analýze programov s dostupným zdrojovým kódom sú ale k dispozícii ďalšie možnosti. Môžu sa odchytiť funkcie priamo v ňom alebo preložiť program pomocou GCC s parametrom `-finstrument-functions` [8].

Bez zdrojového kódu je potrebné modifikovať binárny kód, či už modifikovať všetky monitorované funkcie, alebo upraviť inštrukcie volania funkcií [8]. Technika vkladania inštrukcií do zdrojového kódu a monitorovania rôznych aspektov programu sa nazýva *inštrumentácia*. Môže prebiehať na reálnom stroji, v sandboxe alebo aj emulovanom stroji. Populárnym nástrojom je Intel Pin[4]. Nástroj Valgrind[5] taktiež využíva inštrumentáciu, je robustnejší, ale pomalší oproti Pin-u, ktorý využíva *just-in-time* preklad [16].

Ďalšou možnosťou dynamickej analýzy je využiť debugger. Vložením breakpoint-ov pred volania funkcií môžeme zastaviť beh programu keď dosiahne dané miesto, analyzovať stav CPU, obsah pamäte a odkrokovať si ďalší beh programu. Získané informácie nemusia byť úplné alebo sa netýkajú škodlivých častí kódu, pretože program môže odhaliť spustenie v debuggeri a tak sa vyhnúť analýze.

[4] http://www.intel.com/software/pintool
[5] http://valgrind.org/

Kapitola 3

Clusty

Táto kapitola vychádza z [15].

V AVG Technologies, tak ako aj v iných antivírusových firmách, sa denne analyzuje veľké množstvo vzoriek. Takáto práca sa nedá zvládnuť manuálne, teda vyžaduje si pomoc od automatizovaných nástrojov. *Clusty* je nástroj pre zhlukovú analýzu súborov. Je určený na prvotnú analýzu prichádzajúcich vzoriek. Cieľom je analyzovať vzorky, zistiť vzájomné podobnosti a rozdeliť ich do zhlukov. Tieto informácie sa potom vhodne reprezentujú analytickému tímu, čo by malo urýchliť ich prácu. Zamedzí sa tak ručnej/polo-automatickej analýze vzoriek po jednom kuse.

Podporované sú najčastejšie sa vyskytujúce súborové formáty, ako PE, .NET, ELF, Mach-O. Zhlukovanie využíva prevažne statické informácie získavané z interných nástrojov vyvíjaných v AVG a taktiež z webovej služby VirusTotal[1].

Clusty je implementovaný v jazyku Python s využitím *MongoDB* – multiplatformná NoSQL databáza, ktorá na rozdiel od klasických relačných databáz s tabuľkami ukladá štruktúrované dáta ako JSON dokumenty s dynamickými schémami [24]. Časť nástroja týkajúca sa priamo zhlukovej analýzy je z dôvodu veľkej pamäťovej a časovej náročnosti implementovaná v C++.

3.1 Postup zhlukovej analýzy

V prvej fáze sa analyzujú všetky vzorky:

- zistí sa formát súboru
- podporované formáty sa ďalej analyzujú špecifickými nástrojmi – nástroj fileinfo vyvíjaný v AVG Technologies poskytuje statickú analýzu PE, ELF, Mach-O súborov. Dynamické informácie o PE vzorkách sa získavajú z VirusTotal.
- pre všetky formáty, aj nepodporované, sa získajú informácie nezávislé na formáte
  - detekcie z VirusTotal – informácie o detekciách z iných antivírusových nástrojov
  - obecné informácie o súbore – veľkosť, hash, fuzzy hash (implementovaný v programe ssdeep[2])
- extrahované informácie sa uložia do databázy

[1] https://www.virustotal.com/
[2] http://ssdeep.sourceforge.net

Získané informácie sú uložené priamo v databáze bez nejakých úprav (nevytvára sa z nich žiaden vektor charakteristík), takže informácie o vzorke sú v čitateľnej podobe. Jedinou úpravou je odfiltrovanie niektorých nepodstatných informácii o vzorke, napríklad komunikácia s adresou `127.0.0.1` , `www.google.com` a pod. Ide o bežné informácie, ktoré sa môžu vyskytovať medzi všetkými vzorkami. Zhluky vytvorené na základe takejto spoločnej vlastnosti by mohli obsahovať veľmi odlišné vzorky.

Vzorky, pre ktoré nie je zatiaľ implementovaná špecifická analýza, sú zaradené do kategórie *others*. Zhluková analýza tejto skupiny využíva len obecné informácie o súboroch, napríklad veľkosť, fuzzy hash alebo detekcie z VirusTotal.

V druhej fáze prebieha samotné zhlukovanie. Získané informácie sú už modifikované za účelom znížiť pamäťovú náročnosť (typicky ide o zahashovanie reťazcov) a následne všetky vzorky daného formátu sa algoritmom DBSCAN (viď sekcia 2.3) rozdelia do zhlukov s určitým minimálnym počtom vzoriek a vzdialenosťou (ϵ).

Pre danú skupinu súborov rovnakého formátu sa zhlukovanie spúšťa niekoľkokrát. Pri každom spustení sa zhlukuje len na základe jednej charakteristiky (atribútu). V ďalšom spustení (podľa ďalšej charakteristiky) sa zhlukujú vzorky, ktoré sa nezaradili do žiadneho zhluku pri predchádzajúcich behoch. Najprv sa používajú charakteristiky, ktoré dávajú dobré výsledky. Vhodnosť jednotlivých atribútov a poradie ich použitia vychádza z experimentov. U niektorých atribútov je vyžadovaná presná zhoda, u iných postačuje len podobnosť.

Na záver sa pre každý zhluk vyráta podobnosť jednotlivých atribútov a uložia sa výsledky do databázy. U každého zhluku je potom vidieť, ktoré atribúty, a v akej miere sú spoločné. Spĺňa sa tým požiadavka na použiteľnosť a zrozumiteľnosť výsledkov.

Atribúty vzoriek sú často množiny hodnôt. Pre výpočet podobnosti dvoch množín sa používa algoritmus *Jaccard index* (Jaccard similarity coeficient), viď sekcia 2.2.

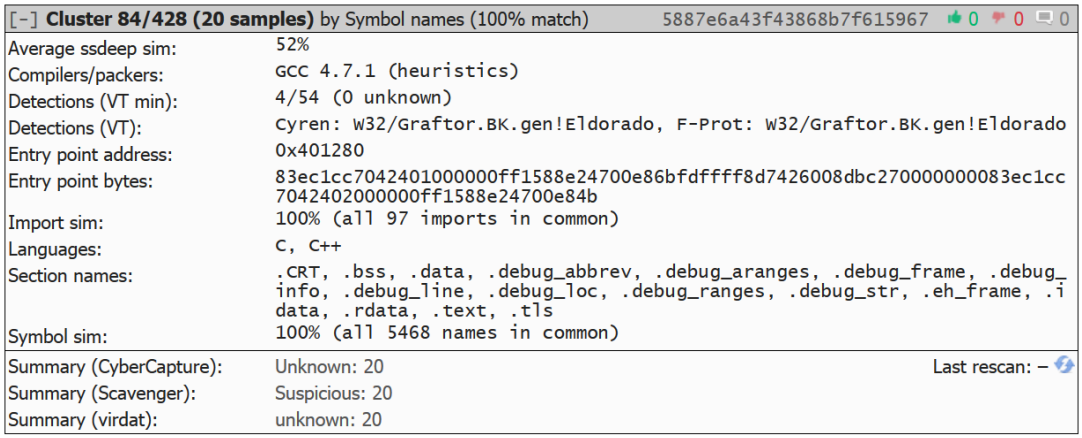
Nástroj je postavený tak, aby zvládol niekoľko stotisíc súborov, čo odpovedá dennému množstvu prichádzajúcich vzoriek. Keďže vzorky prichádzajú postupne počas dňa, tak ich analýza prebieha postupne celý deň. Na záver sa spustí zhluková analýza. To platí pre každodenný beh. Samozrejme, pri spustení nástroja nad nejakým adresárom prebehne analýza všetkých vzoriek v ňom a následne zhlukovanie.

Pri každodennej analýze sa berú do úvahy len vzorky daného dňa. Nie je tam žiadna náväznosť so zhlukmi z predchádzajúcich dní. Nevýhodou teda je, že stále vznikajú nové zhluky a analytici ich musia analyzovať odznova. Ak by dokázal Clusty nájsť podobnosť medzi novým zhlukom a nejakým zhlukom z predchádzajúcich dní, tak by to mohlo rozšíriť informácie o danom zhluku (za predpokladu, že starý zhluk už analytici zanalyzovali). Dôvodom tohto nedostatku je hlavne zložitosť a veľké nároky na výpočtové prostriedky.

3.2 Reprezentácia zhlukov

Pre zobrazenie výsledkov zhlukovej analýzy je vytvorené webové rozhranie. Príklad vytvoreného zhluku je na obrázku 3.1. Ide o skupinu dvadsiatich PE vzoriek. Tento zhluk vznikol na základe rovnakých symbolických mien. Okrem toho majú dané vzorky spoločnú adresu vstupného bodu (angl. *entry point*), 97 rovnakých importov, názvy sekcií a detekcie z VirusTotal. V dolnej časti sú zobrazené súhrnné informácie (položky Summary), ako boli vzorky klasifikované jednotlivými nástrojmi (ide o nástroje z AVG Technologies a Avast).

Informácie o detekciách sa môžu časom zmeniť. Ak si analytik prezerá zhluk vytvorený pred pár dňami, tak detekcie už môžu byť zastaralé. Preto je v pravej dolnej časti tlačidlo,



[-] **Cluster 84/428 (20 samples)** by Symbol names (100% match) 5887e6a43f43868b7f615967 0 0 0

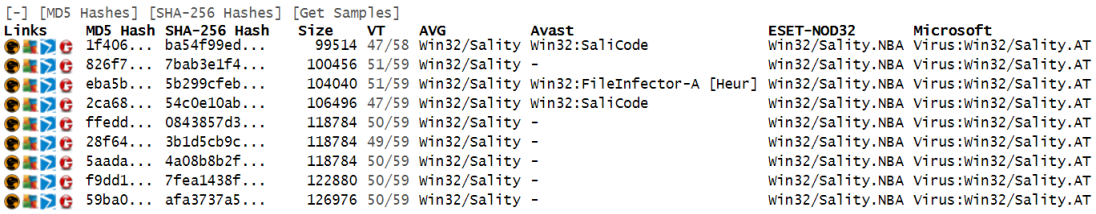
Average ssdeep sim:	52%
Compilers/packers:	GCC 4.7.1 (heuristics)
Detections (VT min):	4/54 (0 unknown)
Detections (VT):	Cyren: W32/Graftor.BK.gen!Eldorado, F-Prot: W32/Graftor.BK.gen!Eldorado
Entry point address:	0x401280
Entry point bytes:	83ec1cc7042401000000ff1588e24700e86bfdffff8d7426008dbc270000000083ec1cc7042402000000ff1588e24700e84b
Import sim:	100% (all 97 imports in common)
Languages:	C, C++
Section names:	.CRT, .bss, .data, .debug_abbrev, .debug_aranges, .debug_frame, .debug_info, .debug_line, .debug_loc, .debug_ranges, .debug_str, .eh_frame, .idata, .rdata, .text, .tls
Symbol sim:	100% (all 5468 names in common)
Summary (CyberCapture):	Unknown: 20 Last rescan: –
Summary (Scavenger):	Suspicious: 20
Summary (virdat):	unknown: 20

Obr. 3.1: Ukážka zhluku PE súborov vytvoreného nástrojom Clusty

ktoré umožňuje aktualizáciu detekcií pre daný cluster (neprebieha už opätovné zhlukovanie ani analýza).

Pre každý zhluk je možné zobraziť si zoznam vzoriek, ako je na obrázku 3.2. Ten obsahuje hash vzorky, jej veľkosť, počet označení za škodlivý program na VirusTotal (VT stĺpec) a detekcie podľa niektorých anitvírusov. Odkazy v ľavom hornom rohu umožňujú stiahnuť si zoznam hashov vzoriek, prípadne tlačidlo `Get samples` stiahne skript na získanie vzoriek samotných. Odkazy v časti Links vyhľadajú danú vzorku v databáze Avast, AVG a na VirusTotal.

Vzorky, ktoré neboli zaradené do žiadneho zhluku sú zobrazené v sekcii *Unclustered*.



[-] [MD5 Hashes] [SHA-256 Hashes] [Get Samples]

Links	MD5 Hash	SHA-256 Hash	Size	VT	AVG	Avast	ESET-NOD32	Microsoft
	1f406...	ba54f99ed...	99514	47/58	Win32/Sality	Win32:SaliCode	Win32/Sality.NBA	Virus:Win32/Sality.AT
	826f7...	7bab3e1f4...	100456	51/59	Win32/Sality	-	Win32/Sality.NBA	Virus:Win32/Sality.AT
	eba5b...	5b299cfeb...	104040	51/59	Win32/Sality	Win32:FileInfector-A [Heur]	Win32/Sality.NBA	Virus:Win32/Sality.AT
	2ca68...	54c0e10ab...	106496	47/59	Win32/Sality	Win32:SaliCode	Win32/Sality.NBA	Virus:Win32/Sality.AT
	ffedd...	0843857d3...	118784	50/59	Win32/Sality	-	Win32/Sality.NBA	Virus:Win32/Sality.AT
	28f64...	3b1d5cb9c...	118784	49/59	Win32/Sality	-	Win32/Sality.NBA	Virus:Win32/Sality.AT
	5aada...	4a08b8b2f...	118784	50/59	Win32/Sality	-	Win32/Sality.NBA	Virus:Win32/Sality.AT
	f9dd1...	7fea1438f...	122880	50/59	Win32/Sality	-	Win32/Sality.NBA	Virus:Win32/Sality.AT
	59ba0...	afa3737a5...	126976	50/59	Win32/Sality	-	Win32/Sality.NBA	Virus:Win32/Sality.AT

Obr. 3.2: Zobrazenie vzoriek konkrétneho zhluku

Kapitola 4

APK formát

APK – *Android application package* je súborový formát založený na formáte ZIP s adresárovou štruktúrou podobnou formátu JAR (*Java Archive*) [18]. JAR je taktiež založený na formáte ZIP, ale obsahuje navyše manifest a prípadne súbory s podpismi [17].

APK formát je využívaný na distribúciu a inštaláciu aplikácii na platforme Android. Každá aplikácia je preložená do jedného APK súboru (prípona `.apk`), ktorý obsahuje bajtkód, zdroje (obrázky, texty apod.), manifest, podpis a ďalšie potrebné súbory. Nasledujúci popis sa zameriava prevažne na obsah APK súboru, pretože na vytvorenie heuristík pre statickú zhlukovú analýzu je potrebné poznať štruktúru a zloženie aplikácií.

4.1 Štruktúra APK

Obsah APK archívu sa líši s každou aplikáciou. Typicky sa ale skladá z nasledujúcich častí:

AndroidManifest.xml

Povinný súbor každej aplikácie s presne takýmto názvom, uložený v koreňovom adresári archívu. Je to XML súbor uložený v binárnej podobe. Poskytuje dôležité informácie, ktoré systém potrebuje pri inštalácii a spúšťaní aplikácie. Patrí sem napríklad názov aplikácie, minimálna Android API verzia, aktivity, služby, požadované povolenia, ale aj povolenia pre iné aplikácie pomocou služieb (angl. *services*) aplikácie [12].

Požadované oprávnenie musí užívateľ schváliť pri inštalácii aplikácie. S Android API 23 bol pridaný ďalší typ oprávnení. V XML je to element `uses-permission-sdk-23`. Tieto oprávnenia sa na zariadeniach s API 23 a vyšším vyžadujú až za behu aplikácie [4].

classes.dex

Ide o súbor vo formáte DEX a obsahuje bajtkód samotnej aplikácie. Podrobnejšie je popísaný v sekcii 4.2.

lib/

Adresár využívaný v natívnych alebo hybridných aplikáciách pre uloženie natívnych knižníc. Typicky sú napísané v jazyku C alebo C++ a ide o výpočtovo náročné alebo časovo kritické časti aplikácie [26]. Môže obsahovať podadresáre pre rôzne architektúry ako armeabi, armeabi-v7a, x86_64, ...

res/

Adresár obsahujúci prostriedky aplikácie – obrázky, textové reťazce, štýly, XML súbory popisujúce menu, animácie a pod. Môžu byť preložené do binárneho formátu. Adresárová štruktúra je obmedzená na podadresáre `anim`, `drawable`, `layout`, `menu`, `values`, `xml` a `raw` (zaručuje uloženie bez prekladu do binárnej podoby).

resources.arsc

Binárny súbor obsahujúci mená a identifikátory zdrojov z adresára `res/`. V zdrojovom kóde sú všetky tieto zdroje prístupné v triede `R.java`, ktorá je vygenerovaná automaticky.

assets/

Adresár obsahujúci nekompilované, nekomprimované súbory. Ide o adresár s nešpecifikovaným účelom, obsahom a adresárovou štruktúrou. Jeho využitie je na autorovi aplikácie. Je bez podpory prostriedkov a identifikácie ako je to pomocou triedy `R.java` pre adresár `res/`. Prístup k týmto súborom je možný pomocou API triedy `AssetManager` [12].

META-INF/

Adresár obsahujúci súbory, ktoré zaručujú integritu ostatných súborov v archíve. Obsahuje tri povinné súbory:

- `MANIFEST`.MF – obsahuje názov a hash všetkých zdrojových súborov (teda okrem tohto adresára).
- `CERT`.SF – zoznam súborov spolu s ich SHA1 hashmi z `MANIFEST.MF` a jeden hash vytvorený z týchto hashov.
- `CERT`.RSA – certifikát s verejným kľúčom.

Každá aplikácia musí byť podpísaná asymetrickým kľúčom RSA alebo DSA. Podpis nemusí byť overený certifikačnou autoritou (tzv. *self-signed* certifikát). Využívajú sa hlavne na zabezpečenie integrity a dôveryhodnosti. Android umožňuje aktualizáciu aplikácie, len ak sa zhoduje certifikát. Navyše, aplikácie s rovnakým certifikátom môžu byť spustené v jednom procese, čo prináša väčšiu modularitu. Aplikácia s iným certifikátom sa nainštaluje ako nová aplikácia. Týmto sa zabráni podvrhnutiu škodlivej verzie aplikácie (tzv. *man in the middle* útok) [19].

4.2 DEX formát

DEX je súborový formát obsahujúci bajtkód, ktorý je určený pre spustenie vo virtuálnom stroji *Dalvik*. Vytvorenie tohto súboru je typicky následovné:

- Zdrojový kód je napísaný v jazyku Java.
- `.java` súbory sú preložené do `.class` súborov – bajtkód pre virtuálny stroj pre Javu (JVM).
- Tento bajtkód je preložený do bajtkódu pre Dalvik. Kód zo všetkých zdrojových súborov je spojený do jedného DEX súboru (okrem špeciálneho prípadu popísaného v ďalšej sekcii).

Existujú aj reverzné nástroje pre tento formát. Známy disassembler je baksmali[1]. Nástroj dex2jar[2] ide ešte vyššie a prekladá bajtkód z DEX do JAR formátu. Spätným prekladačom pre Javu tak môžeme získať zdrojový kód priamo v Jave.

Štruktúra DEX formátu

Štruktúra DEX formátu vo verzii 035 a 037[3] (prevzatá z [1]) je nasledovná:

- hlavička – prvých osem bajtov tvorí signatúra (`dex\n035\0` alebo `dex\n037\0` pridaná od Android 7.0), ďalej obsahuje kontrolný súčet adler32 pre celý súbor okrem časti so signatúrou a samotným kontrolným súčtom, veľkosť celého súboru a hlavičky, endianita, veľkosti a polohy ďalších polí nasledujúcich za hlavičkou.
- reťazce – zoradený list všetkých reťazcov použitých v tomto súbore – teda nie len reťazce zo zdrojového kódu, ale aj názvy tried, typov atď. Identifikátory z nasledujúcich častí obsahujú ukazatele na tieto reťazce.
- typy – identifikátory všetkých použitých typov (triedy, atribúty tried, primitívne typy)
- identifikátory prototypov metód
- identifikátory atribútov tried
- identifikátory metód
- definície tried – obsahuje informácie o nadtriede, prístupové práva (`public`, `private`, `final`, ...), názov zdrojového súboru, implementované rozhrania, anotácie, počiatočné hodnoty statických atribútov, popisy atribútov a metód – identifikátor, prístupové práva, kód
- dáta – potrebné pre polia uvedené vyššie, napríklad kód metód, anotácie
- linkované dáta – využité v staticky linkovaných súboroch

DEX súbor teda poskytuje dostatok informácii na spätné zrekonštruovanie tried, ich metód, atribútov, parametrov, dátových typov, aj s pôvodnými názvami.

Celkový počet metód, na ktoré sa dá odkazovať v rámci DEX súboru je ale obmedzený na 65 536, pretože veľkosť odkazu je len dva bajty. Zahŕňa to Android metódy, metódy z knižníc a vlastný kód. Označuje sa to ako 64 K reference limit. V starších verziách operačného systému Android až po verziu 4.4.4 sa spúšťali aplikácie vo virtuálnom stroji Dalvik. Tým bola aplikácia obmedzená len na jeden DEX súbor. S využitím *multidex* knižnice, ktorá musí byť súčasťou súboru `classes.dex` je možné pristúpiť k bajtkódu ďalších DEX súborov.

S príchodom Android-u 5.0 (API level 21) bol virtuálny stroj Dalvik nahradený novým behovým prostredím *Android Runtime* – ART. Pri inštalácií aplikácie sa prekladá bajtkód do natívnych inštrukcií, do `.oat` súboru. Vyhľadajú a preložia sa všetky `classesN.dex` súbory. Odpadá tak nutnosť použiť mutlidex knižnicu [2]. Viac o ART je v [1, 29]. Obsah APK archívu sa ale nijako nemení, stále je tam bajtkód v DEX formáte.

[1] https://github.com/JesusFreke/smali/wiki

[2] https://github.com/pxb1988/dex2jar

[3] Kvôli nešpecifikovanej chybe bola verzia 036 vynechaná (https://web.archive.org/web/20161119053607/https://source.android.com/devices/tech/dalvik/dex-format.html)

4.3 ZIP formát

ZIP je rozšírený súborový formát pre kompresiu a archiváciu súborov. Jeho autorom je Phil Katz. Prvá verzia bola predstavená v roku 1989 spoločnosťou PKWARE v rámci balíka *PKZIP* [5]. Kompresia súborov je bezstratová a podporuje niekoľko rôznych komprimovacích algoritmov (napr. UnShrinking, Imploding, Tokenizing, BZIP2, LZMA, Deflating). Každý súbor archívu môže byť uložený komprimovaný alebo bez kompresie (metóda označovaná ako stored), zašifrovaný alebo digitálne podpísaný, bez ohľadu na spôsob uloženia ostatných súborov. Integrita je zabezpečená kontrolným súčtom CRC32 pre každý súbor [20].

4.3.1 Štruktúra archívu

V archíve môžu byť umiestnené ľubovoľné typy súborov, aj iné ZIP archívy. Jednotlivé súbory sú popísané metadátami obsahujúcimi potrebné informácie pre ich spravovanie. Každý typ týchto dát začína dvojicou bajtov `PK` – `0x4b50`.

Základné rozloženie archívu je na obrázku 4.1. Na najvyššej úrovni ho môžeme rozdeliť do troch hlavných častí:

- metadáta a dáta súborov – jeden záznam pre každý súbor. Lokálna hlavička súboru začína signatúrou `0x04034b50`. Obsahuje informácie o verzii ZIP formátu potrebnej na extrahovanie súboru, dva bajty príznakov, typ kompresie, časové razítko, kontrolný súčet CRC32, komprimovanú a reálnu veľkosť súboru, názov súboru a extra položku pre uloženie dodatočných informácii (napríklad informácie špecifické pre daný operačný systém).

  Položka *popis dát* je využitá len v prípade nastaveného tretieho bitu v poli príznakov. V takomto prípade sú hodnoty CRC32, komprimovaná a reálna veľkosť súboru v lokálnej hlavičke vynulované a uložené v tejto položke (má doporučenú, nie povinnú, signatúru `0x08074b50`). Je uložená hneď za dátami súboru a využíva sa v prípade, že nie je možné meniť ľubovoľne pozíciu vo výslednom ZIP archíve, teda nebolo možné dané informácie zapísať do lokálnej hlavičky.

  Šifrovacia hlavička je využitá len v prípade ochrany súboru heslom.

- centrálny adresár – obsahuje jednu hlavičku centrálneho adresára pre každý súbor a začína štvoricou bajtov `0x02014b50`. Väčšina informácií je rovnakých ako v lokálnej hlavičke odpovedajúceho súboru. Navyše je tam verzia formátu ZIP, ktorou bola táto hlavička vytvorená, interné a externé atribúty, a môže tam byť uvedený komentár k súboru. Dôležitou položkou je poloha lokálnej hlavičky daného súboru.

  Pred týmito hlavičkami sú ešte umiestnené položky dešifrovacia hlavička a extra dáta. Boli pridané vo verzii formátu 6.2 a súvisia s funkciou šifrovania obsahu hlavičiek centrálneho adresára.

- koniec centrálneho adresára – začína signatúrou `0x06054b50`. Obsahuje informácie o umiestnení a veľkosti centrálneho adresára a počet položiek v ňom. Navyše tam môže byť umiestnený komentár.

lokálna hlavička súboru 1
hlavička pre šifrovanie 1
dáta súboru 1
popis dát 1
...
lokálna hlavička súboru n
hlavička pre šifrovanie n
dáta súboru n
popis dát n
dešifrovacia hlavička
extra dáta
hlavička centrálneho adresára 1
...
hlavička centrálneho adresára n
koniec centrálneho adresára

Tab. 4.1: Štruktúra ZIP archívu (prevzatá z [20])

Vo verzii 4.5 bola pridaná podpora pre ZIP64 rozšírene. Vo vyššie popísanom rozložení archívu pribudli nové časti – ZIP64 koniec centrálneho adresára a ZIP64 lokátor pre túto položku. Niektoré položky môžu využiť rozšírenie zo štyroch na osem bajtov. Podrobnejšie je to popísané v [5].

4.3.2 Zneužívanie ZIP formátu

Napriek presne definovanému formátu sa rôzne nástroje odlišujú v spôsobe otvárania archívu. Niektoré sú tolerantné k chybných archívom, iné nie. Napríklad nástroj `unzip v6.0` na Linuxe nedokáže otvoriť archív bez jeho poslednej časti – koniec centrálneho adresára. Na druhej strane, nástroj `jar`[4] (The Java Archive Tool) otvorí daný archív bez problémov a nepotrebuje k tomu centrálny adresár ani koniec centrálneho adresára. Lineárne prechádza zhora nadol lokálne hlavičky súborov a extrahuje dáta.

Tieto rôzne implementácie prinášajú možnosti zneužitia. Autori vírusov to môžu využívať a tým sťažovať analýzu, najmä automatizovanú analýzu.

Jednoduchý spôsob ako znemožniť otvorenie ZIP súboru pre niektoré nástroje je využiť komentár na konci archívu. Koniec centrálneho adresára obsahuje veľkosť komentára na dvoch bajtoch. To limituje dĺžku komentára na 65 535 znakov. Python implementácia modulu `zipfile` sa spolieha na túto dĺžku a v prípade dlhšieho komentára nenájde koniec centrálneho adresára. Rovnako zlyhá aj `unzip`, ale `jar` extrahuje súbory z daného archívu bez problémov.

Ďalším dobrým príkladom je implementácia otvárania ZIP archívov pri inštalácii APK na Androide. APK formát podporuje len dva spôsoby uloženia súboru – `stored` a kompromáciu metódou `deflate` [18]. Ako je uvedené v ukážke kódu 4.1, otestuje sa, či ide o súbor komprimovaný metódou `stored` (`if` vetva), ak nie, tak sa automaticky dedukuje použitie metódy `deflate` (`else` vetva). To umožňuje autorovi vytvoriť ZIP (APK), v ktorom bude daný súbor naozaj uložený metódou `deflate`, ale v hlavičke bude uvedená napríklad metóda `shrunk` alebo nejaké dva bajty, rôzne od metód `stored` a `deflate`. Týmto sa APK podarí

[4] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html

nainštalovať na zariadenie s Androidom a zároveň sa skomplikuje, prípadne až znemožní analýza daného súboru v archíve pre väčšinu nástrojov (napr. pre antivírusový skener). Podrobnejšie je daná praktika popísaná v [18]. Týka sa operačného systému Android vo verzii 4.4 a nižšie. Je to už síce staršia verzia, ale používa sa ešte stále asi na 12 % zariadení s Androidom [3].

```
if (method == kCompressStored) {
    if (sysCopyFileToFile(fd, pArchive->mFd, uncompLen) != 0)
        goto bail;
} else {
    if (inflateToFile(fd, pArchive->mFd, uncompLen, compLen) != 0)
        goto bail;
}
```

Kód 4.1: Ukážka otvárania súboru z archívu v Android module `ziparchive.cpp` vo verzii 4.4

Kapitola 5

Návrh

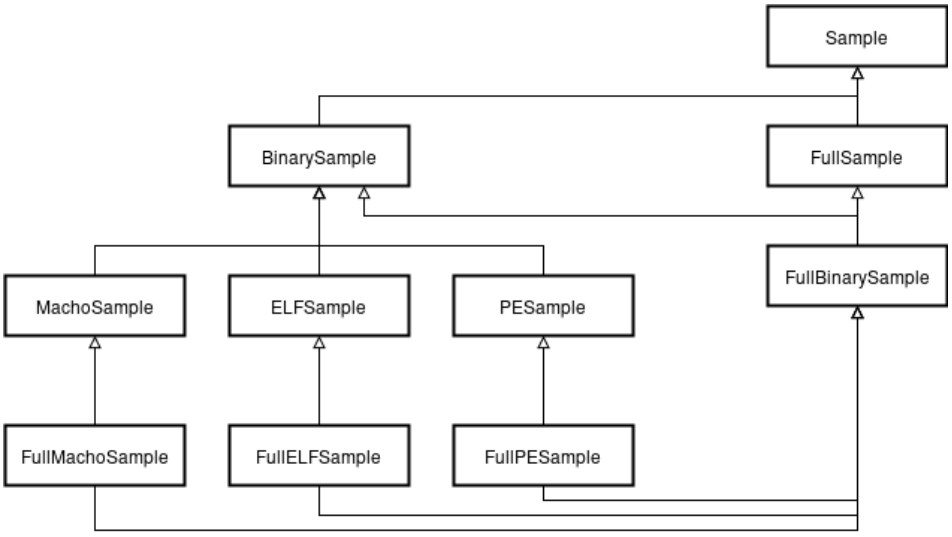
Pridávanie podpory pre zhlukovú analýzu ďalších súborových formátov môžeme rozdeliť na dve časti. Najprv je potrebné rozšíriť časť nástroja Clusty napísanú v jazyku Python, ktorá analyzuje súbory a získané súbory vhodne reprezentuje a ukladá do databázy. Druhá časť sa týka samotnej zhlukovej analýzy implementovanej v C++. Tam sa musia doplniť metódy na zhlukovanie súborov podľa vybraných charakteristík. Dôležitou úlohou je výber tých najvhodnejších. Analýza, či už statická alebo dynamická, môže poskytnúť veľké množstvo informácií o súbore. Z praktického hľadiska (nároky na hardvér) nie je vhodné využívať všetko pri obrovskom množstve vzoriek, taktiež nie všetky atribúty prinášajú žiadané výsledky.

Na analýzu súborov väčšiny formátov Clusty využíva externé nástroje. Získané informácie sa následne pre každú vzorku uložia do objektu triedy `Sample`, presnejšie do podtriedy reprezentujúcej daný formát. Trieda `FullSample` sa stará o výber správnej podtriedy. Má k dispozícii jednotlivé analyzátory. Jedným z nich je aj trieda `FileAnalyzer`. Pomocou nej sa identifikuje typ a podtyp daného súboru, napríklad typ – binárny súbor, podtyp – PE. Následne na to určeným analyzátorom analyzuje tento súbor a vybrané informácie uloží do danej triedy (`FullPESample` v tomto prípade). Vzťah medzi triedami `Sample` a `FullPESample` je zobrazený v diagrame 5.1. Obdobne je to spravené aj pre ďalšie typy súborov – textové súbory, skripty, atď.

Diagram 5.1 zobrazuje hierarchiu tried reprezentujúcich binárne súbory, konkrétne PE, ELF a Mach-O. Všetky tieto súborové formáty obsahujú binárny kód, preto majú spoločnú nadtriedu `BinarySample` a `FullBinarySample`, kde sa ukladajú informácie špecifické pre všetky binárne súbory. V triede `FullSample` sú uložené informácie nezávislé na formáte. Informácie sú uložené v týchto triedach tak, ako boli získané z analýzy, prípadne s menšími úpravami. Jedna z nich je odfiltrovanie niektorých informácií nepotrebných (alebo zavádzajúcich) pre zhlukovú analýzu, napríklad odstránenie IP adresy `127.0.0.1`, viď strana 15.

Informácie, s ktorými pracuje C++ nástroj sú tie, ktoré boli do databázy uložené z tried v Python časti. Preto je hierarchia tried v C++ podobná. Je ale mierne zjednodušená, neobsahuje triedy s prefixom `Full`. Dôležitým rozdielom týchto tried od tried v Python časti je spôsob reprezentovania informácii. Väčšina informácii o vzorkách je množina reťazcov (napr. volania API funkcií). Z dôvodu optimalizácie sú reťazce načítané z databázy nahradené hashmi. Výnimkou sú situácie, kedy je týchto reťazcov málo a chceme uchovať túto informáciu v atribútoch vytvoreného zhluku. To umožní zobraziť dané reťazce tak, ako bolo ukázané na obrázku 3.1.

Ďalším dôvodom používania `FullSample` v Python časti oproti `Sample` v C++ je, že C++ nástroj sa používa len na zhlukovanie. V Python časti nemôžu byť informácie určené



Obr. 5.1: Triedy reprezentujúce informácie o PE, ELF a Mach-O súboroch v Python časti

na zhlukovanie v rámci triedy `Sample`, pretože napríklad jej podtrieda `WebSample` (reprezentácia vzoriek pre zobrazenie na webe) uchováva mierne odlišné informácie.

V nasledujúcej sekcii je popísaný návrh rozšírenia Clusty-ho o ďalšie formáty – APK, archívy a DEX. Dôvodom pre rozšírenie o archívy je, že APK je tiež archív. Preto je pridaná táto základná podpora.

DEX súbory ako také sa prakticky nepoužívajú, vždy sú súčasťou APK. Vírusová databáza v AVG Technologies ale dostáva aj samotné DEX vzorky, a preto je pridaná aj podpora pre zhlukovú analýzu tohto formátu.

5.1 Rozšírenie nástroja o APK formát

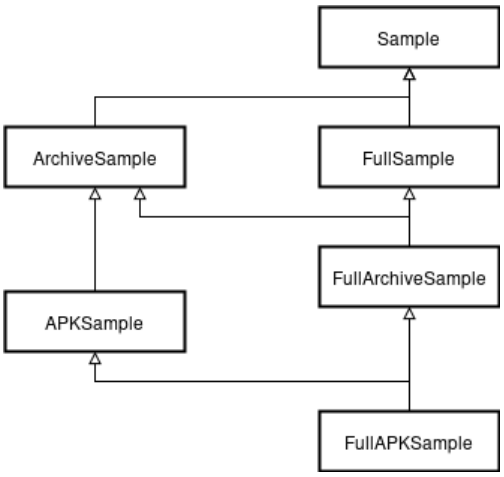
Popísaná štruktúra reprezentovania informácií umožňuje jednoduché rozšírenie nástroja o ďalšie súborové formáty. Najprv je ale potrebné detekovať APK súbor. Typ a podtyp súboru sa určuje v triede `FileAnalyzer`, viď strana 23. APK súbor je v podstate ZIP archív so špecifickým obsahom. Dôležité súbory sú `AndroidManifest.xml` a `classes.dex`. Prítomnosť aspoň jedného z nich v archíve teda môže naznačovať, že ide o APK súbor. Táto heuristika sa využije v triede `FileAnalyze` na detekciu APK. Výsledný typ pre APK súbor teda bude *archív* a podtyp bude *APK*.

APK je komplexný archív a vyžaduje si vlastný analyzátor. Ten sa pridá k ostatným analyzátorom, ktoré už `FullSample` využíva. Pre uloženie získaných informácii z analýzy je potrebné pridať ďalšiu podtriedu triedy `Sample`.

Naivným prístupom postačí pridať triedu `APKSample` (dedí zo `Sample`) a `FullAPKSample` (dedí z FullSample) a ukladanie APK informácií je hotové. Lenže APK súbor je v podstate ZIP archív a v prípade rozširovania nástroja o zhlukovú analýzu ZIP archívov by sme mohli mať sčasti duplicitnú implementáciu pre APK a pre archívy. Preto je výsledný návrh zapojenia APK reprezentácie rozšírený o triedy pre archívy, ako je to zobrazené na diagrame 5.2.

Takto sú informácie o APK týkajúce sa každého archívu (napríklad cesty k súborom) v triede `FullArchiveSample`. Zároveň je nástroj schopný reprezentovať aj samotné archívy,

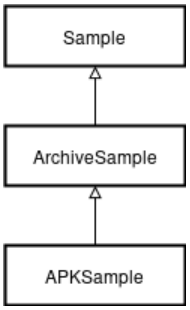
nie len APK. Teda rozšírenie o podporu RAR, TAR, 7-ZIP a ďalších archívov bude môcť priamo využívať tieto triedy.



Obr. 5.2: Triedy reprezentujúce informácie o archívoch a APK súboroch v Python časti

Rozšírenie o APK formát v C++ časti je podobné ako rozšírenie v Python časti. Informácie týkajúce sa obecne archívov sú reprezentované triedou `ArchiveSample` a špecifické informácie o APK sú uložené v podtriede `APKSample` (viď diagram 5.3).

Navyše je potrebné ešte pridať zhlukovacie metódy pre jednotlivé APK charakteristiky a metódy na výpočet podobností vzoriek vo vytvorených zhlukoch.

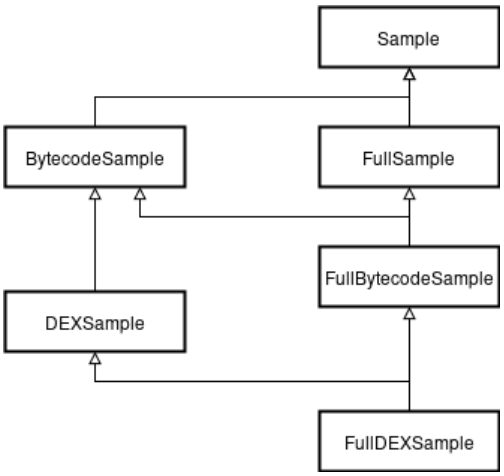


Obr. 5.3: Triedy reprezentujúce informácie určené pre zhlukovanie o archívoch a APK súboroch v C++ časti

5.2 Rozšírenie nástroja o DEX formát

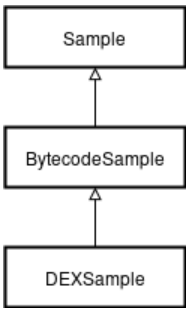
Detekcia DEX súborov v triede `FileAnalyzer` je jednoduchšia než APK. Tieto súbory začínajú vždy bajtmi `dex\n035\0` (037 od verzie Android 7.0, viď sekcia 4.2).

DEX súbory obsahujú bajtkód pre virtuálny stroj Dalvik. Takže ich môžeme obecne označiť za bajtkód a využiť podobnú hierarchiu tried ako pre APK. Informácie obecne platné pre bajtkód sú reprezentované triedou `FullBytecodeSample` a atribúty typické len pre DEX súbor sú uložené v podtriede `FullDEXSample`, ako je to zobrazené v diagrame 5.4.



Obr. 5.4: Triedy reprezentujúce informácie o DEX súboroch v Python časti

Podobne ako to bolo už pre APK, je do C++ časti pridaná reprezentácia bajtkódu – `BytecodeSample`. Jej podtrieda `DEXSample` je určená pre informácie o DEX súboroch, ako zobrazuje diagram 5.5.



Obr. 5.5: Triedy reprezentujúce informácie určené pre zhlukovanie o DEX súboroch v C++ časti

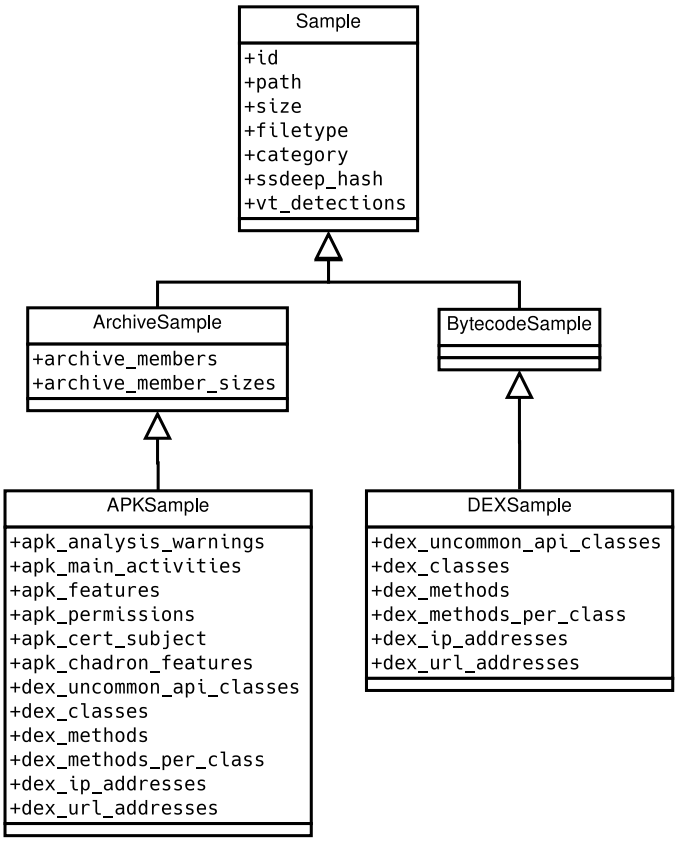
Podrobnejší diagram APK a DEX tried v C++ nástroji je v ukážke 5.6.

Navyše je potrebné ešte pridať zhlukovacie metódy pre jednotlivé DEX charakteristiky a metódy na výpočet podobností vzoriek vo vytvorených zhlukoch.

5.3 Výber charakteristík pre zhlukovanie

APK ako súborový formát je veľmi komplexný a obsahuje súbory rôznych formátov. O každom jeho súbore by sa dalo zistiť veľké množstvo informácii. Pri zhlukovej analýze je ale dôležité vybrať tie správne charakteristiky. Dôvodom je získanie kvalitných výsledkov, ale aj pamäťová a časová náročnosť, keďže nástroj by mal zvládnuť spracovať niekoľko desiatok aj stoviek tisíc vzoriek. Z nasledujúcich charakteristík sa dá každá určitým spôsobom obísť, preto sa nedá spoľahnúť len na jednu z nich.

Na najvyššej úrovni APK formátu, teda ak sa naň pozrieme ako na archív, môžeme využiť tieto informácie:



Obr. 5.6: Diagram APK a DEX tried v C++ nástroji

- cesty k súborom
- veľkosti súborov – ich výhoda je odolnosť proti obfuskovaniu názvov súborov, ale môže spojiť aj dva úplne odlišné súbory

Ďalšie vlastnosti sa dajú už získať z jednotlivých súborov archívu:

- `AndroidManifest.xml` – medzi najdôležitejšie informácie v tomto súbore patria určite oprávnenia. Ďalej sú tam uvedené požiadavky na hardvér a softvér (*uses-feature* položka), hlavné aktivity a vstupné body aplikácie.

  Z požadovaných oprávnení sa môže ešte zúžiť výber na nebezpečné oprávnenia[1] alebo na oprávnenia požadované až za behu aplikácie, viď sekcia 4.1.

  Okrem oprávnení sa väčšina informácii (názov balíčka, aktivity, . . .) dá veľmi ľahko obfuskovať premenovaním.

- `classes.dex` – tento súbor obsahuje dostatok informácií z pôvodného Java kódu. Dajú sa z neho získať celé triedy, ich metódy, parametre, atribúty. Ďalej obsahuje reťazce, z ktorých sa dajú využiť informácie o použitých Android API triedach, IP a URL adresy alebo e-maily.

[1] https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous

Samotný bajtkód sa dá taktiež využiť na zhlukovanie. Porovnávaniu aplikácii pomocou bajtkódu (len inštrukcie, bez operandov) sa venuje [32]. Bajtkód rozdeľuje na kratšie úseky, ktoré zahashuje pomocou fuzzy hashu a tie potom porovnáva.

Porovnávanie bajtkódu metódou *n-grams* využíva DroidKin [10].

Všetky tieto informácie sú reťazce a ľahko sa dajú zmeniť alebo skryť. Populárnym nástrojom na obfuskovanie APK súborov je ProGuard[2] a nevyužívajú ho len autori vírusov. Aj bežné aplikácie ho používajú ako obranu proti reverznému inžinierstvu.

Ďalším problémom aj pri bajtkóde môže byť využívanie knižníc tretích strán, ktoré môžu vyvolať zhodu pre rôzne aplikácie.

- `META-INF/CERT.RSA` – certifikát obsahuje verejný kľúč, informácie o autorovi, platnosť certifikátu. Problémom je, že nemusí byť overený certifikačnou autoritou a tak môže mať každá aplikácia iný podpis, napriek tomu, že patria do rovnakej rodiny.
- obrázky, audio a video súbory – tieto súbory sa môžu ľahko meniť naprieč aplikáciami. Relevantnou informáciou by mohla byť ikona aplikácie.
- ELF súbory – často sa už v aplikácii vyskytujú tieto súbory, väčšinou z dôvodu rýchlosti, ale môže sa v nich vyskytovať aj škodlivý kód.

Popísané charakteristiky sa dajú získať statickou analýzou APK. Dynamická analýza môže niektoré statické informácie spresniť, napríklad použitie konkrétnych IP a URL adries a volanie API funkcií. Taktiež môže priniesť informácie o vstupno–výstupných operáciách, prístupe k úložisku dát, atď.

Sandbox Chadron vyvíjaný v AVG Technologies poskytuje dynamickú analýzu APK súborov. Výstupom analýzy je vektor zachytených charakteristík obsahujúci napríklad volanie API funkcií, prístup k dátovému úložisku, sieťová komunikácia, požadované oprávnenia. Zároveň sa snaží pomocou strojového učenia a získaných charakteristík rozhodnúť o škodlivosti daného súboru. V ukážke 5.7 sú zobrazené zachytené charakteristiky. Je tam volanie API metódy, žiadosť o povolenie k zápisu dát, prístup na úložisko dát a sieťová komunikácia.

```
Time            ID      Name              Detail
1467624487066   0x1001  sendTextMessage   private void android.telephony.SmsManager().
    sendTextMessageInternal((java.lang.String)"124", (java.lang.String)"456", (java.
    lang.String)"ahoj", (android.app.PendingIntent)null, (android.app.PendingIntent)
    null, (boolean)true)

1487710788483   0x50de  pd_android_permission_WRITE_MEDIA_STORAGE
1492869519209   0x600b  ioWriteStorage    /storage/emulated/0/Android/data/com.adups.
    fota/files/crash/crash-2017-04-22-15-58-39-1492869519216.txt

1467624487076   0x1007  getDeviceId       public java.lang.String android.telephony.
    TelephonyManager().getDeviceId()

1492523790132   0x600f  httpRequest       10.0.2.15:57952 -> www.a.shifen.com:80
```

Kód 5.7: Ukážka charakteristík zachytených v sandboxe Chadron

[2]https://www.guardsquare.com/en/proguard

5.4 Výber nástroja na analýzu APK

Existuje veľa nástrojov a knižníc na statickú, dynamickú, online analýzu alebo reverzné inžinierstvo APK súborov. Pre účely tejto práce je potrebné analyzovať hlavne DEX súbory a `AndroidManifest.xml` súbor. Výber bol zúžený na niekoľko najpopulárnejších a stále udržiavaných:

- Apktool[3] – je to robustný nástroj napísaný v jazyku Java určený pre reverzné inžinierstvo APK súborov. Ponúka disassembler, dekódovanie binárnych XML súborov v rámci APK
- ApkParser[4] – ide o nástroj napísaný v Jave, ale neextrahuje dostatok potrebných informácii z APK ani DEX súborov.
- Androguard[5] – tento nástroj napísaný v jazyku Python 2 poskytuje analýzu DEX, APK, binárnych XML v rámci APK, disassembler a spätný prekladač. Jeho API[6] poskytuje jednoduché rozhranie pre vytváranie ďalších nástrojov na analýzu týchto súborových formátov.
- RAPID[7] – ďalší Java nástroj poskytujúci API na analýzu DEX súborov. Neposkytuje ale žiadnu analýzu manifestu `AndroidManifest.xml`.

K ďalším nástrojom môžme zaradiť disassemblery (napríklad smali/baksmali[8], dex2jar, dedexer[9]) a spätné prekladače z Dalvik bajtkódu do Java bajtkódu, napríklad dex2jar, Enjarify[10]. Spätnými prekladačmi Java bajtkódu je možné získať až zdrojový kód v jazyku Java. Nevýhodou týchto reverzných nástrojov je nutnosť vytvoriť ďalší parser nad ich výstupom, či už ide o strojový kód alebo Java kód.

Z popísaných nástrojov bol pre analýzu APK a DEX súborov zvolený Androguard. Jeho API poskytuje jednoduchý prístup k množstvu informácií, ktoré sa dajú získať priamo z DEX súboru a `AndroidManifest.xml` súboru. Nie je potrebné vytvárať parser bajtkódu alebo Java kódu a nehrozí chybná analýza spôsobená spätným prekladom. Ďalšou jeho výhodou je implementačný jazyk Python, aj keď ide o verziu 2 a Clusty je napísaný vo verzii 3.

[3] https://ibotpeaches.github.io/Apktool/
[4] https://github.com/clearthesky/apk-parser
[5] https://github.com/androguard/androguard
[6] http://doc.androguard.re/html/index.html
[7] https://github.com/unhcfreg/RAPID
[8] https://github.com/JesusFreke/smali
[9] http://dedexer.sourceforge.net/
[10] https://github.com/google/enjarify

Kapitola 6

Implementácia

V tejto kapitole je popísaná implementácia rozšírenia nástroja Clusty, navrhnutá v kapitole 5, získavanie statických informácii z APK pomocou nástroja Androguard a prístup k dynamickým informáciám zo sandboxu Chadron. Na úvod je popísaná implementácia podpory archívov, keďže APK je archív a rozšírenie nástroja bolo navrhnuté aj so základnou podporou pre archívy.

Python časť nástroja Clusty má za úlohu analyzovať vzorky a uložiť získané informácie do databázy. Implementácia je určená pre Python verziu 3.4 a dokumentácia je generovaná nástrojom Sphinx[1]. Pre správu modulov tretích strán sa používa virtuálne prostredie vytvorené nástrojom virtualenv[2].

Zhluková analýza je implementovaná v jazyku C++, konkrétne vo verzii C++ 14. Pre preklad sa používa nástroj CMake s prekladačom GCC a dokumentácia je generovaná nástrojom doxygen[3].

Na správu revízii sa používa verzovací nástroj Git.

6.1 Analýza archívov

Charakteristiky vybrané pre zhlukovú analýzu archívov v kapitole 5.3 sú platné obecne pre archívy, netýkajú sa konkrétne ZIP formátu alebo TAR. Ide len o cesty k súborom a skutočné veľkosti súborov. Tieto informácie sa dajú získať z Python modulov pre prácu s danými archívmi. Ide o tieto moduly:

- `zipfile` – modul zo štandardnej knižnice pre ZIP formát
- `tarfile` – modul zo štandardnej knižnice pre TAR formát
- `rarfile` – modul z PyPI[4] pre RAR formát
- `libarchive-c` – modul z PyPI, wrapper nad C knižnicou `libarchive`, zvolelný na analýzu 7-ZIP archívov

Triedy zapuzdrujúce prácu s týmito štyrmi archívnymi formátmi sú implementované v module `clusty.analyses.archive`. Informácie, ktoré sa z nich získavajú, sú obecné

[1] http://www.sphinx-doc.org/en/stable/
[2] https://pypi.python.org/pypi/virtualenv
[3] http://www.stack.nl/~dimitri/doxygen/
[4] Repozitár Python knižníc tretích strán (https://pypi.python.org/pypi)

pre archívy, takže trieda `FullArchiveSample`, ktorá ukladá tieto informácie, nepotrebuje poznať konkrétny typ. Preto je v triede `FileAnalyzer` implementovaná metóda (kontextový manažér) `open_archive`, ktorá sa postará o analýzu archívu danou triedou. Kód tejto metódy je zobrazený v ukážke 6.1. V prípade, že daný archív nie je podporovaný, vráti sa objekt triedy `NoArchive`. Ide o implementáciu návrhového vzoru *Null object*.

```
@contextlib.contextmanager
def open_archive(self, archive, filetype=None):
try:
    try:
        if self._is_zip_archive(filetype):
            arch = ZipArchive(archive)
        elif self._is_rar_archive(filetype):
            arch = RarArchive(archive)
        elif self._is_tar_archive(filetype):
            arch = TarArchive(archive)
        elif self._is_7zip_archive(filetype):
            arch = Zip7Archive(archive)
        else:
            arch = NoArchive()
    except ArchiveReadError:
        arch = NoArchive()
    yield arch
finally:
    arch.close()
```

Kód 6.1: Kontextový manažér otvárania archívu v triede `FileAnalyzer`

APK ako ZIP archív sa otvára pomocou modulu `zipfile`. Tento modul vyžaduje striktné dodržiavanie ZIP formátu, inak vyvolá výnimku a analýza skončí. Nie všetky vzorky, ktoré prichádzajú na analýzu, dodržiavajú pravidlá tohto formátu. Najčastejšie prípady porušenia archívu sú:

- komentár dlhší než povolený limit
- chýbajúca časť *koniec centrálneho adresára* prípadne aj *centrálny adresár*
- je nastavený príznak použitia UTF-8 reťazca pre názov súboru, ale názov nie je platný UTF-8 reťazec
- chybný kontrolný súčet CRC32 u nejakého súboru

Ako bolo popísané v sekcii 4.3.2, nie všetky nástroje majú s týmito chybami problém. Nástroj jar často dokáže extrahovať aj takéto archívy. Jeho použitie by znamenalo rozbaľovanie a opätovné vytváranie chybných archívov. Nevýhodou je, že je to externý nástroj, archívy sa musia rozbaľovať a je tam riziko otvorenia tzv. *zip bomb* archívu.

Cieľom nástroja Clusty je analyzovať aj takéto súbory a získať aspoň zopár informácii, ktoré môžu pomôcť pri hľadaní zhlukov. V triede `clusty.analyses.archive.ZipArchive` je implementovaná oprava týchto chýb (okrem nesprávneho CRC32).

Pri porušení dĺžky komentára je oprava jednoduchá. Stačí hľadať signatúru konca centrálneho adresára od konca archívu. `zipfile` modul to robí tiež takto, ale prehľadáva len posledných 64 kB, čo je formátom definovaná maximálna dĺžka komentára. Ak sa signatúra nepodarí nájsť alebo ani po jej nájdení nevznikne korektný archív, tak sa predpokladá neprítomnosť sekcií centrálny adresár a koniec centrálneho adresára.

Chýbajúce časti archívu sa opravujú zložitejšie. Je potrebné zrekonštruovať centrálny adresár a koniec centrálneho adresára tak, aby výsledný archív odpovedal formátu. Ako bolo

popísané v sekcii 4.3.1, väčšina metadát v hlavičkách centrálneho adresára je rovnakých ako v lokálnych hlavičkách súborov. Postup rekonštrukcie je teda takýto:

- lineárne sa prechádza archív od začiatku a hľadá sa signatúra lokálnej hlavičky súboru
- pre každú lokálnu hlavičku sa vytvorí jedna hlavička v centrálnom adresári týmto spôsobom:
  - metadáta, ktoré sú spoločné v lokálnej aj centrálnej hlavičke, sa použijú pre vytvorenie novej centrálnej hlavičky
  - niektoré metadáta sa nedajú získať a musia byť v centrálnej hlavičke vynulované
  - doplní sa správna pozícia lokálnej hlavičky
- zrekonštruovaný centrálny adresár sa pripojí na koniec archívu
- na záver sa pripojí koniec centrálneho adresára. Ten obsahuje počet položiek v centrálnom adresári a jeho polohu. Tieto informácie sú k dispozícii, keďže centrálny adresár bol skonštruovaný v predchádzajúcich krokoch.

Je možné, že nie všetky súbory archívu sa dajú po oprave otvoriť. Často je obsah posledného súboru skrátený, ale to sa už nedá zrekonštruovať.

Problému s názvom reťazca, ktorý by mal byť v UTF-8 kódovaní, ale nie je, sa dá taktiež vyhnúť. Jedenásty bit v poli príznakov (viď sekcia 4.3.1) ovplyvňuje kódovanie. V prípade jeho nastavenia na hodnotu 1 musí byť názov súboru a komentár v UTF-8 kódovaní. Vynulovaním tohto bitu sa použije kódovanie IBM CP 437 [20]. Po tejto úprave archívu sa už nedá spoľahnúť na korektnosť názvu súboru, ale je možné ho otvoriť a analyzovať, čo je dôležitejšie pre nástroj Clusty.

Pri pokuse o otvorenie súboru s chybným CRC súčtom vyvolá Python modul `zipfile` výnimku. Toto správanie môže byť v určitých situáciách žiadané a odpovedá ZIP formátu. Pre nástroj Clusty je ale užitočnejšie analyzovať takýto súbor a získať informácie vhodné pre zhlukovú analýzu. Trieda `zipfile.ZipFile` neponúka žiadnu možnosť, ako ignorovať kontrolu CRC. Dá sa to ale obísť, viď kód 6.2 (tzv. *monkey patch*). Tento kód funguje korektne pre Python verziu 2.7 na Linuxe.

```
import zipfile
zipfile.ZipExtFile._update_crc = lambda *args, **kwargs: None
```

Kód 6.2: Patch metódy kontrolujúcej CRC súčet v Python `zipfile` module

6.2 Analýza APK a DEX

Androguard je robustný nástroj poskytujúci statickú analýzu APK, disassembling, aj spätný preklad DEX súborov. Čo ale neposkytuje, je vhodná reprezentácia charakteristík potrebných pre zhlukovanie.

S využitím jeho API sa môže jednoducho vytvoriť vlastný nástroj. Konkrétne je možné na analýzu APK a DEX využiť tieto dve triedy[5]:

- `androguard.core.bytecodes.apk.APK`

[5] http://doc.androguard.re/html/index.html

- `androguard.core.bytecodes.dvm.DalvikVMFormat`

Analýza pomocou tohto API nemôže byť priamo zapojená do nástroja Clusty, pretože Androguard je napísaný v jazyku Python vo verzii 2. Preto sú vytvorené dva samostatné nástroje v tejto verzii, analyzátor DEX súborov (`analyze_dex.py`) a analyzátor APK súborov (`analyze_apk.py`), ktorý využíva aj ten prvý analyzátor pre analýzu svojich DEX súborov. Tieto nástroje sa pridajú k ďalším externým nástrojom, ktoré využíva Clusty. Informácie získané z Androguard API sú v týchto skriptoch ešte ďalej spracované a reprezentované vo formáte JSON. Spracovanie sa týka konkrétne reťazcov obsiahnutých v DEX súbore. Androguard poskytuje len zoznam týchto reťazcov, ale v `analyze_dex.py` sa z reťazcov získavajú Android API triedy, IP adresy a URL adresy.

Androguard používa pre otváranie ZIP archívov Python modul `zipfile`. Takže má tiež problém s otváraním archívov neodpovedajúcich ZIP formátu. Preto je potrebné mať na vstupe archívy, ktoré už boli opravené, ako je popísané v sekcii 6.1.

V samotnej implementácii Androguard tried sa vyskytujú chyby, ktoré znemožňujú analýzu. Vo väčšine prípadov to končí výnimkou a teda nie sú poskytnuté žiadne informácie o analyzovanom súbore. Tieto chyby som opravil priamo v Androguard implementácii:

- práca s `None` objektom v `APK.get_main_activity()`
- prístup na neplatný index v `APK.decodeLength()`
- prístup k neexistujúcej položke slovníka v `DalvikVMFormat.get_type_list()`
- odchytenie výnimky pri chybe pri otváraní súboru z archívu
- zrušenie nekontrolovaného vypisovania informácii na štandardný chybový výstup. Je pridaná možnosti zapnúť tieto výpisy pomocou Python modulu `logging`.
- detekcia konca súboru – chybné porovnávanie načítaných bajtov a veľkosti súboru
- rozšírenie analýzy oprávnení v AndroidManifest.xml súbore a pridanie metódy na získanie oprávnení (`APK.get_runtime_permissions()`), ktoré je možné na zariadení s Android API 23 a vyšším vyžadovať až za behu aplikácie, viď sekcia 4.1.
- optimalizácia prístupu k súboru `AndroidManifest.xml` v archíve otvorenom pomocou modulu `zipfile`

O analýzu APK a DEX súborov sa v nástroji Clusty stará trieda `APKAnalyzer` v module `clusty.analyses.apk`. Pomocou Python modulu `subprocess` zavolá daný analyzátor a výstup vo formáte JSON použije na vytvorenie objektu triedy `clusty.analyses.APKInfo` (`clusty.analyses.DEXInfo` pre DEX súbory). Tento objekt sa potom použije pre naplnenie informácii do `FullAPKSample` (`FullDEXSample` pre DEX), ako bolo uvedené v 5.1.

Volanie programu pomocou `subprocess` modulu ilustruje[6] kód 6.3. Na treťom riadku kódu by mal byť uvedený proces, ktorý chceme spustiť. Je tam ale `pypy`, a to nie je skript na analýzu APK. APK analyzátor je Python skript a je uvedený až na ďalšom riadku.

PyPy[7] je alternatívna implementácia k CPython interpretu pre verziu 2.7. Hlavnou výhodou je rýchlosť a pamäťová náročnosť. Zrýchlenie prináša ale len pre procesy bežiace aspoň pár sekúnd, pretože používa just-in-time prekladač [21].

[6] V skutočnosti je na volanie podprocesu použitý Clusty modul `cmd_runner`, ale je to len wrapper nad `subprocess.Popen`.

[7] https://pypy.org

```
p = subprocess.Popen(
    [
        'pypy',
        self._analyze_apk_path,
        file
    ],
    stderr=subprocess.PIPE,
    stdout=subprocess.PIPE
)
stdout, stderr = p.communicate(input, timeout)
```

Kód 6.3: Spustenie procesu na analýzu APK súbor v triede `APKAnalyzer`

Analýza DEX súborov v Androguarde často trvá niekoľko sekúnd, pri väčších súboroch (pár megabajtov) sú to aj desiatky sekúnd. Dôvodom je prístup k informáciám o triedach a ich metódach, v prípade, že ich je aj niekoľko tisíc (knižnice tretích strán, napríklad pre zobrazovanie reklamy, mapy).

Dynamická analýza APK

Na dynamickú analýzu APK súborov je využitý sandbox Chadron vyvíjaný v AVG Technologies. Výstupom jeho analýzy je vektor zachytených charakteristík obsahujúci napríklad volanie API funkcií, prístup k dátovému úložisku, sieťová komunikácia, požadované oprávnenia. Zároveň sa snaží pomocou strojového učenia a získaných charakteristík, rozhodnúť o škodlivosti daného súboru. Tieto informácie sú uložené v PostgreSQL databáze. Chadron má aj webové rozhranie pre prístup k získaným informáciám a výsledkom strojového učenia.

Prístup k výsledkom dynamickej analýzy je implementovaný v module `clusty.chadron`. Trieda `Chadron` implementuje získavanie informácií z webového rozhrania pomocou Python modulu `requests`[8]. Ide o ďalší analyzátor, ktorý má trieda `FullSample` k dispozícii a využíva ho len na analýzu APK vzoriek. Výsledkom volania metódy `Chadron.analyze()` je trieda `ChadronSampleInfo` reprezentujúca informácie získané dynamickou analýzou – vektor charakteristík.

Výsledok strojové učenia, teda či je vzorka škodlivá, alebo nie, sa nepoužíva pri zhlukovej analýze. Slúži len ako súhrnná informácia pre zhluk, zobrazujúca počet škodlivých/podozrivých/neškodných vzoriek v danom zhluku, viď príloha B.1, položka *Summary (Chadron)*.

6.3 Zhluková analýza

Zhluková analýza súborov je implementovaná v jazyku C++.

Z databázy sa načítajú získané informácie o súboroch a použijú sa na vytvorenie odpovedajúcich tried. Trieda `APKSample` reprezentuje APK súbory, `DEXSample` je pre DEX súbory a `ArchiveSample` obsahuje informácie o archívoch, ako bolo popísané v kapitole 5.

Implementácia heuristík a metód pre záverečný výpočet spoločných vlastností vzoriek v zhlukoch je podobná pre tieto tri formáty, a preto bude následujúci popis zameraný len na APK formát. Metódy pre ďalšie formáty sa líšia prevažne len výberom charakteristík.

[8] http://docs.python-requests.org/en/master/

Zhlukovacie metódy

Implementácia týchto metód využíva pre zhlukovanie algoritmus DBSCAN (viď sekcia 2.3). Tieto metódy vždy pracujú s jednou charakteristikou. Vyrátajú pre ňu maticu podobnosti (viď sekcia 2.1) a nad ňou sa spúšťa DBSCAN. Táto časť už je v nástroji Clusty implementovaná.

Rozšírenie o podporu ďalších formátov si vyžaduje implementáciu metód, ktoré budú rátať vzdialenosť pre jednotlivé atribúty vzoriek. Tieto metódy sa nachádzajú v module `clusty/clustering/methods/regular.hpp`. Pre APK formát sú pridané dva typy metód. Metódy, ktoré si vyžadujú presnú zhodu charakteristík (podobnosť nemá zmysel alebo má horšie výsledky), napríklad zhoda v hlavných aktivitách aplikácie. Ukážka je v kóde 6.4. V nástroji Clusty sa používa hodnota 0 pre úplnú zhodu a hodnota 100 pre maximálnu vzdialenosť.

```
template<typename SampleT>
void cluster_samples_by_apk_main_activities_100_match(
      Samples<SampleT>& samples, Clusters& clusters, std::size_t min_cluster_size) {
    cluster_samples_having_property_by(
        samples,
        clusters,
        [](const auto& sample) { return sample->has_apk_main_activities(); },
        [](const auto& s1, const auto& s2) {
            return s1->get_apk_main_activities() ==
                s2->get_apk_main_activities() ? 0 : 100;
        },
        "APK main activities (100% match)",
        min_cluster_size
    );
}
```

Kód 6.4: Výpočet vzdialenosti dvoch vzoriek podľa hlavných aktivít

Druhým typom metód sú metódy, ktoré počítajú vzdialenosť pomocou podobnostných metód. Ukážka tohto výpočtu je v kóde 6.5. Konkrétne sa používa funkcia Jaccard koeficient, viď sekcia 2.2, ale s upraveným rozsahom hodnôt. V nástroji Clusty bol upravený na interval $\langle 0, 100 \rangle$, a v tejto práci bol len prevzatý.

Výnimkou pri výpočte vzdialenosti je fuzzy hash (ssdeep). Tam je algoritmus výpočtu vzdialenosti odlišný a netýka sa tejto práce.

V module `clusty/clustering/categories/apk.hpp` je implementovaná metóda, ktorá dostane na vstupe všetky APK vzorky a pomocou popísaných vzdialenostných metód a algoritmu DBSCAN ich roztriedi do zhlukov. Výber vhodných charakteristík a ich poradie je popísané v kapitole 8.

Na záver zhlukovej analýzy sú pre každý zhluk vyrátané spoločné vlastnosti. Pre atribúty, u ktorých má zmysel ich podobnosť (napríklad názvy tried), sa vyráta podobnosť (v %) naprieč všetkými vzorkami daného zhluku. U atribútov, kde sa vyžaduje presná zhoda (napríklad hlavné aktivity), sa tento atribút označí za spoločnú vlastnosť v prípade, že je rovnaký u všetkých vzoriek zhluku. Ukážka spoločných vlastností zhluku je v prílohe B.1.

```
template<typename SampleT>
void cluster_samples_by_apk_permissions_similarity(
      Samples<SampleT>& samples, Clusters& clusters, std::size_t min_cluster_size) {
    cluster_samples_having_property_by(
        samples,
        clusters,
        [](const auto& sample) { return sample->has_apk_permissions(); },
        [](const auto& s1, const auto& s2) {
            return 100 - jaccard_similarity(
                s1->get_apk_permissions(),
                s2->get_apk_permissions()
            );
        },
        "APK␣permissions␣(similarity)",
        min_cluster_size
    );
}
```

Kód 6.5: Výpočet vzdialenosti dvoch vzoriek podľa požadovaných oprávnení

Kapitola 7

Testovanie

Počas práce prebiehalo testovanie na dvoch úrovniach. Prvá časť sa venuje testovaniu implementácie Python a C++ časti. Ďalšie testovanie sa týka výsledkov zhlukovej analýzy a kvality vytvorených zhlukov.

Testovanie implementácie v Python časti

Implementácia bola dôsledne testovaná jednotkovými a integračnými testmi. V Python časti nástroja bol využitý testovací modul `unittest`. Pokrytie kódu jednotkovými a integračnými testmi je v tabuľke 7.1. Vypísaný zoznam ale neobsahuje všetky súbory implementované v rámci tejto práce. Nie sú tam samotné testy a ani skripty `analyze_apk.py` a `analyze_dex.py`, pretože priamo pre ne neboli vytvorené testy. Ich výstup sa ale používa v integračných testoch, kedy sa testujú reálne vzorky a takto sú nepriamo otestované aj tieto skripty.

Súbor	Pokrytie riadkov	Bez pokrytia	Pokrytie
`clusty/analyses/apk.py`	114	0	100 %
`clusty/analyses/archive.py`	223	16	93 %
`clusty/chadron.py`	85	0	100 %
`clusty/samples/apk_sample.py`	2	0	100 %
`clusty/samples/archive_sample.py`	2	0	100 %
`clusty/samples/bytecode_sample.py`	2	0	100 %
`clusty/samples/dex_sample.py`	2	0	100 %
`clusty/samples/full_apk_sample.py`	55	0	100 %
`clusty/samples/full_archive_sample.py`	11	0	100 %
`clusty/samples/full_bytecode_sample.py`	4	0	100 %
`clusty/samples/full_dex_sample.py`	34	0	100 %

Tab. 7.1: Pokrytie kódu jednotkovými a integračnými testmi v Python časti

Na spúšťanie testov je využitý nástroj nose[1]. Kód 7.1 zobrazuje spustenie jednotkových testov. Celkovo ich bolo v tejto práci pridaných 86. Sekvenčná doba behu týchto testov je len 0.2 sekundy. Počet zobrazených pokrytých riadkov je nižší, než je v skutočnosti v zdrojových súboroch, pretože tento nástroj ich ráta iným spôsobom.

Integračných testov bolo pridaných 75. Ich sekvenčné spustenie trvá asi 110 sekúnd, viď ukážka kódu 7.2.

[1] https://nose.readthedocs.io/en/latest/

V prípade zlyhania testu je vypísaný tzv. *traceback* s detailnejším popisom chyby, viď ukážka 7.3. Je tam uvedený konkrétny súbor, číslo riadku a aj objekty, ktorých porovnávanie zlyhalo.

```
$ nosetests tests/unit/
.....................................................................................
....
----------------------------------------------------------------------
Ran 86 tests in 0.211s

OK
```

Kód 7.1: Spustenie jednotkových testov pomocou nástroja nose

```
$ nosetests tests/real/
.....................................................................
----------------------------------------------------------------------
Ran 75 tests in 109.557s

OK
```

Kód 7.2: Spustenie integračných testov pomocou nástroja nose

```
$ nosetests tests/unit/analyses/apk_tests.py
.........................F..
======================================================================
FAIL: test_permissions_returns_list_of_permissions (tests.unit.analyses.apk_tests.
    APKInfoTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pavol/clusty-pp-bp/tests/unit/analyses/apk_tests.py", line 152, in
      test_permissions_returns_list_of_permissions
      self.assertEqual(info.permissions, [])
      AssertionError: Lists differ: ['android.send_sms'] != []

      First list contains 1 additional elements.
      First extra element 0:
      'android.send_sms'

      - ['android.send_sms']
      + []

      ----------------------------------------------------------------------
      Ran 31 tests in 0.013s

      FAILED (failures=1)
```

Kód 7.3: Výpis pri zlyhaní testu

Testovanie implementácie v C++ časti

Zhluková analýza implementovaná v C++ je testovaná jednotkovými testmi pomocou nástroja Google Test[2]. Výsledky pokrytia kódu sú zobrazené v tabuľke 7.2. Testujú sa tam

[2]https://github.com/google/googletest

moduly reprezentujúce informácie o jednotlivých súboroch a zhlukovacie metódy pre jednotlivé kategórie.

Súbor	Pokrytie riadkov		Funkcie	
src/clusty/samples/apk_sample.cpp	100 %	16 / 16	100 %	4 / 4
src/clusty/samples/apk_sample.hpp	100 %	49 / 49	100 %	25 / 25
src/clusty/samples/archive_sample.cpp	100 %	6 / 6	100 %	4 / 4
src/clusty/samples/archive_sample.hpp	100 %	11 / 11	100 %	6 / 6
src/clusty/samples/bytecode_sample.cpp	100 %	5 / 5	100 %	4 / 4
src/clusty/samples/bytecode_sample.hpp	100 %	1 / 1	100 %	1 / 1
src/clusty/samples/dex_sample.cpp	100 %	10 / 10	100 %	4 / 4
src/clusty/samples/dex_sample.hpp	100 %	25 / 25	100 %	13 / 13
src/clusty/clustering/categories/apk.hpp	100 %	35 / 35	100 %	1 / 1
src/clusty/clustering/categories/archive.hpp	100 %	19 / 19	100 %	1 / 1
src/clusty/clustering/categories/dex.hpp	100 %	19 / 19	100 %	1 / 1
src/clusty/clustering/methods/regular.cpp	97 %	38 / 39	100 %	10 / 10
src/clusty/clustering/methods/regular.hpp	60 %	146 / 243	56 %	87 / 153

Tab. 7.2: Pokrytie kódu jednotkovými testmi v C++ časti

Ukážku spustenia testov zobrazuje kód 7.4. Celkovo bolo pridaných 53 testov. Sekvenčná doba ich behu je len 26 ms.

```
$ ./build/tests/clusty/clusty_tests
...
[----------] 2 tests from have_same_prefix_up_to_tests
[ RUN      ] have_same_prefix_up_to_tests.
    returns_true_when_they_have_same_prefix_up_to
[       OK ] have_same_prefix_up_to_tests.
    returns_true_when_they_have_same_prefix_up_to (0 ms)
[ RUN      ] have_same_prefix_up_to_tests.
    returns_false_when_they_do_not_have_same_prefix_up_to
[       OK ] have_same_prefix_up_to_tests.
    returns_false_when_they_do_not_have_same_prefix_up_to (0 ms)
[----------] 2 tests from have_same_prefix_up_to_tests (0 ms total)

[----------] Global test environment tear-down
[==========] 53 tests from 9 test cases ran. (26 ms total)
[  PASSED  ] 53 tests.
```

Kód 7.4: Spustenie Google Tests testov

Testovanie vytvorených zhlukov

Pre účel testovania kvality vytvorených zhlukov boli využité známe vzorky, ktoré už sú rozdelené do rodín. Kontrola kvality zhluku prebiehala manuálne. Sledovala sa podobnosť jednotlivých charakteristík v rámci zhluku a rodiny vzoriek v rámci zhluku. Dôležitejšia metrika pre vytvorený zhluk je rodina vzoriek daného zhluku. V ideálnom prípade by mali byť v danom zhluku vzorky len jednej rodiny a daná rodina by mala byť rozdelená na čo najmenší počet zhlukov. Samozrejme, nie všetky vírusy danej rodiny sú postavené na rovnakej implementácii. Môžu sa odlišovať, takže z hľadiska zhlukovej analýzy je ich rozdelenie do viacerých zhlukov akceptovateľné.

Pre každú charakteristiku sa spustila zhluková analýza 12 000 vzoriek s už definovanými rodinami. Počet a veľkosti vytvorených zhlukov sa porovnávali s inými charakteristikami.

Taktiež sa sledovalo množstvo zhlukov so vzorkami z rôznych rodín. Ukážka porovnania výsledkov zhlukovej analýzy podľa vybraných charakteristík je v tabuľke 7.3. Podľa počtu zhlukov a zaradených vzoriek má skvelé výsledky atribút oprávnenia. Kvalita zhlukov je ale dôležitejšia, a v tom jednoznačne prevažuje počet metód na triedu a názvy tried. Pri použití názvov tried je ale celkový počet vzoriek v zhlukoch menší. Kvalita zhlukov pri zhlukovej analýze podľa veľkostí súborov v archíve a podľa hlavných aktivít je podobná, ale pri aktivitách vznikali výrazne menšie zhluky. Teda jedna rodina bola rozdelená na malé časti.

Charakteristika	**Počet zhlukov**	**Celkový počet vzoriek v zhlukoch**	**Kvalita zhlukov**
Počet metód na triedu	233	9310	Zhluky tvorené vzorkami z jednej rodiny, mix rodín je veľmi ojedinelý.
Podobnosť názvov tried	150	4 777	Veľmi malý výskyt rôznych rodín v rámci zhluku.
Podobnosť veľkostí súborov v archíve	173	8 109	Občas je v zhluku zopár vzoriek z inej rodiny.
Podobnosť API tried	191	10 550	Častejší výskyt dvoch aj viac rodín v rámci zhluku.
Zhoda hlavných aktivít	153	4 389	Občas je v zhluku zopár vzoriek z inej rodiny. Vznikajú menšie zhluky.
Podobnosť oprávnení	204	10 185	Častý výskyt dvoch, niekedy viacerých rodín v rámci zhluku.

Tab. 7.3: Porovnanie kvality zhlukov vytvorených zhlukovou analýzou 12 000 vzoriek

Kapitola 8

Zhodnotenie

Všetky rozšírenia nástroja Clusty navrhnuté v kapitole 5 sú implementované tak ako bolo popísané v kapitole 6. Práca je už nasadená do produkčnej verzie. Každodenná zhluková analýza prichádzajúcich vzoriek je už teda rozšírená o formáty APK, DEX a archívy (ZIP, TAR, RAR a 7-ZIP).

Denne sa musí zvládnuť analyzovať a roztriediť desiatky tisíc APK vzoriek. DEX formát a archívy majú menšie zastúpenie. Štatistiky pamäťovej a časovej náročnosti pre vyše 20 000 APK súborov sú v ukážke kódu 8.1. Doba behu analýzy a zhlukovej analýzy je 40 minút a maximálna veľkosť okupovanej pamäte (angl. *Maximum resident set size*) je 2.2 GB. Tieto hodnoty boli namerané na stroji s operačným systémom Debian, k dispozícii bolo 72 jadier procesora a 128 GB operačnej pamäte.

```
$ ls apk-samples/ | wc -l
22713
$ /usr/bin/time -v ./cluster_samples.py apk-samples/
Command being timed: "./cluster_samples.py apk-samples/"
    Elapsed (wall clock) time (h:mm:ss or m:ss): 40:11.57
    Maximum resident set size (kbytes): 2200436
```

Kód 8.1: Časová a pamäťová náročnosť zhlukovej analýzy 22 713 APK vzoriek

K väčším nárokom na pamäť prispieva hlavne používanie knižníc tretích strán, ktoré sú taktiež zahrnuté v bajtkóde v `classes.dex` súbore. Tieto triedy sa filtrujú (tzv. *blacklisting*), ale ich zoznam nebude nikdy kompletný a pri analýze v nástroji Androguard žiadne filtrovanie neprebieha.

Podrobnejší log zhlukovej analýzy týchto vzoriek je v prílohe A.1 a ukážka troch vytvorených zhlukov je v prílohe B.1. 22 713 vzoriek bolo rozdelených do 873 zhlukov a 1073 vzoriek nebolo zaradených do žiadneho zhluku. Tieto čísla sú ovplyvnené parametrami *min_points* a ε algoritmu DBSCAN, viď sekcia 2.3. Minimálny počet vzoriek pre vytvorenie zhluku je nastavený na 5. Tieto parametre už boli experimentálne nastavené pre nástroj Clusty a v rámci tejto práce boli len prevzaté.

V logu je detailnejšie popísaná aj doba zhlukovej analýzy v jednotlivých fázach. Samotná analýza vzoriek trvala 38 minút, teda väčšinu času, a zhlukovanie v C++ časti prebehlo len za 107 sekúnd, teda necelé 2 minúty.

Analýza APK súborov trvá najdlhšie v porovnaní s inými formátmi (PE, ELF, ...). Najpomalšou časťou je Androguard. Výhody jeho použitia ale prevýšili nevýhody a doba analýzy je akceptovateľná. Prispieva k tomu aj to, že každodenný beh nástroja analyzuje

prichádzajúce vzorky postupne počas celého dňa. Výrazné zrýchlenie analýzy prináša použitie interpretu PyPy, viď sekcia 6.2. Porovnanie oproti interpretu CPython je v ukážke kódu 8.2.

```
$ time python2.7 analyze_apk.py sample.apk  # CPython interpret

real    1m1.863s
user    0m59.108s
sys     0m1.460s

$ time pypy analyze_apk.py sample.apk  # PyPy interpret

real    0m9.313s
user    0m6.532s
sys     0m1.200s
```

Kód 8.2: Porovnanie rýchlosti analýzy APK súboru interpretmi CPython a PyPy

Vhodnosť charakteristík pre zhlukovanie

Z APK atribútov popísaných v sekcii 5.3 sú pre zhlukovanie vybrané tieto:

- počet metód na triedu
- názvy tried – je potrebné ale odfiltrovať knižnice tretích strán
- informácie získané z dynamickej analýzy
- veľkosti súborov v archíve
- cesty k súborom archívu – treba sa ale vyhnúť chybným zhodám, ktoré vznikajú v prípade APK súborov obsahujúcich len základné súbory – `classes.dex`, `META-INF/`, `AndroidManifest.xml` a `resources.arsc`.

  Táto charakteristika je úspešná hlavne pri väčšom množstve súborov, typicky v adresároch `assets/`, `res/` a `lib/`.
- vstupné body aplikácie
- oprávnenia na prístup k hardvéru a softvéru
- API triedy – je potrebné ale odfiltrovať bežné triedy ako `android.graphics`, `android.support`, `android.widget` a `android.view`
- IP a URL adresy – bežné IP adresy sú ale od filtrované

Pre archívy sa používajú tieto atribúty:

- veľkosti súborov
- cesty k súborom

U DEX súborov sa používajú tie isté charakteristiky ako boli použité aj pre APK formát, teda:

- počet metód na triedu
- názvy tried
- API triedy
- IP a URL adresy

Okrem týchto atribútov špecifických pre daný formát sa využívajú aj obecné vlastnosti súborov, ako fuzzy hash (ssdeep), veľkosť súboru alebo detekcie z VirusTotal.

Vymenované atribúty sú zoradené zostupne podľa kvality výsledných zhlukov overenej experimentálne.

Tieto charakteristiky navrhnuté v sekcii 5.3 nie sú použité pre zhlukovanie:

- názov balíčka – problémom je obfuskovanie, preto väčšinou vznikali veľmi malé zhluky
- certifikát – certifikát nemusí byť overený certifikačnou autoritou. Vzorky z jednej rodiny teda majú často iný certifikát.
- ELF súbory – často sa využívajú knižnice tretích strán, a preto vznikali zhluky so vzorkami rôznych rodín
- nebezpečné oprávnenia – je to veľmi malá skupina oprávnení a výsledky teda obsahovali často vzorky z rôznych rodín
- ikony – v rámci jednej rodiny sa vyskytuje rôznorodosť ikon, prípadne sa v rôznych rodinách využíva rovnaké Android logo

V rámci tejto práce boli splnené tieto úlohy:

- výber nástroja na analýzu APK a DEX formátov (zvolený bol Androguard)
- výber charakteristík získaných statickou a dynamickou analýzou, ktoré môžu byť vhodné pre zhlukovú analýzu
- vytvorenie skriptov na statickú analýzu. Informácie získané z nástroja Androguard ešte analyzujú podrobnejšie a reprezentujú ich vo formáte JSON pre nástroj Clusty.
- oprava chýb pri analýze v nástroji Androguard a rozšírenie analýzy oprávnení
- získavanie informácii z dynamickej analýzy z nástroja Chadron
- rozšírenie Python časti nástroja o detekciu, analýzu a reprezentáciu týchto formátov
- rozšírenie C++ časti o zhlukovú analýzu týchto súborových formátov
- zhodnotenie jednotlivých charakteristík a výber tých najvhodnejších
- rozšírenie nástroja o zhlukovú analýzu archívov – ZIP, RAR, TAR, 7-ZIP
- implementácia opravy a rekonštrukcie ZIP archívov, ktoré nedodržiavajú ZIP formát. Táto oprava je dôležitá aj pre APK formát, keďže je to taktiež ZIP archív. Zvýši sa tým množstvo analyzovaných vzoriek a aj šanca na zaradenie vzorky do niektorého zhluku.

Kapitola 9

Záver

V tejto práci bola popísaná zhluková analýza, niektoré najznámejšie metódy a jej využitie v antivírusvých firmách pri analýze súborov. Konkrétne bol predstavený nástroj Clusty vyvíjaný v AVG Technologies. Taktiež boli rozobrané súborové formáty APK, DEX a ZIP, keďže cieľom tejto práce bolo rozšíriť tento existujúci nástroj o zhlukovú analýzu Android aplikácii, teda APK súborov. Boli vybrané vhodné charakteristiky a implementované heuristiky. Rozšírenie nástroja o APK, DEX a archívy je už nasadené do produkčnej verzie a tak sa vhodnosť implementovaných heuristík potvrdzuje každodenne.

Hlavným cieľom práce bolo rozšírenie nástroja o súborový formát APK. Všetky body zadania boli splnené a navyše je pridaná aj zhluková analýza pre DEX súbory a archívy. Ďalší vývoj by sa mohol zamerať na vylepšenie heuristík pre APK formát. Zhluková analýza APK a DEX formátu sa môže rozšíriť o porovnávanie vzoriek podľa bajtkódu, ako už bolo navrhnuté v sekcii 5.3.

Pre zhlukovú analýzu archívov bola pridaná len základná podpora a mala by sa využiť pri rozšírení o viacero archívnych formátov. Taktiež sú otvorené možnosti rozšírenia tejto časti o nové heuristiky, keďže archív obsahuje rôzne typy súborov. Ich analýza (napríklad s využitím už existujúcich riešení v nástroji) môže priniesť vhodnejšie charakteristiky pre zhlukovanie archívov.

Využitie viacerých atribútov pre výpočet podobnosti vzoriek by sa dalo aplikovať u všetkých súborových formátov. Aktuálne sa pri každej iterácii počíta podobnosť objektov len podľa jedného atribútu. Niektoré atribúty majú slabšie výsledky a ich vhodnou kombináciou by sa to mohlo zlepšiť. Taktiež sa môžu využiť aj iné typy metód, napríklad hierarchické môžu lepšie odrážať rozdelenie vzoriek do jednotlivých rodín.

Testovanie kvality vytvorených zhlukov by sa mohlo zautomatizovať, keďže sa kontroluje hlavne počet rôznych rodín v rámci zhluku a veľkosti zhlukov. Urýchli sa tým vyhodnocovanie vhodnosti použitia heuristík, ale len za predpokladu, že bude k dispozícii trénovacia sada vzoriek.

Literatúra

[1] Android: *ART and Dalvik*. [Online; cit. 20. 01. 2017].
URL source.android.com/devices/tech/dalvik/

[2] Android: *Configure Apps with Over 64K Methods*. [Online; cit. 20. 01. 2017].
URL developer.android.com/studio/build/multidex.html

[3] Android: *Dashborads*. [Online; cit. 28. 03. 2017].
URL developer.android.com/about/dashboards/index.html

[4] Android: *Requesting Permissions at Run Time*. [Online; cit. 20. 01. 2017].
URL developer.android.com/training/permissions/requesting.html

[5] Barosan, D.: *Study on a known-plaintext attack on ZIP encryption*. 2015, [Online; cit. 10. 01. 2017].
URL www.delaat.net/rp/2014-2015/p57/report.pdf

[6] Bergeron, J.; Debbabi, M.; Erhioui, M. M.; aj.: *Static Analysis of Binary Code to Isolate Malicious Behaviors*. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999. (WET ICE '99) Proceedings. IEEE 8th International Workshops on*, 1999, ISSN 1080-1383, s. 184 – 189.

[7] Cheng, W.; Wang, W.; Batista, S.: *Grid-based clustering*. In *Data Clustering: Algorithms and Applications*, kapitola 6, 2014.
URL cs.unc.edu/~weicheng/chapter.pdf

[8] Egele, M.; Scholte, T.; Kirda, E.; aj.: *A Survey on Automated Dynamic Malware Analysis Techniques and Tools*. In *ACM Computing Surveys (CSUR)*, ročník 44, ACM, 2012, ISSN 0360-0300, s. 1 – 42.

[9] Everitt, B. S.; Landau, S.; Leese, M.; aj.: *Cluster Analysis*. Chichester: Wiley, 2011, ISBN 978-0-470-74991-3, 330 s.

[10] Gonzalez, H.; Stakhanova, N.; Ghorbani, A. A.: DroidKin: Lightweight Detection of Android Apps Similarity. In *International Conference on Security and Privacy in Communication Networks*, Springer, 2014, ISBN 978-3-319-23829-6, s. 436 – 453.

[11] Kinable, J.; Kostakis, O.: *Malware Classification based on Call Graph Clustering*. 2010, [Online; cit. 20. 01. 2017].
URL arxiv.org/pdf/1008.4365.pdf

[12] Komatineni, S.; MacLean, D.: *Pro Android 4*. Apress, 2012, ISBN 978-1-4302-3931-4.

[13] Kovac, P.: *Fighting Malware With GPUs In Real Time*. [Online; cit. 25. 01. 2017].
URL
on-demand.gputechconf.com/gtc/2015/presentation/S5612-Peter-Kovac.pdf

[14] Křoustek, J.: *Retargetable Analysis of Machine Code*. Dizertační práce, Vysoké Učenie Technické v Brne. Fakulta Informačných Technoogií, 2014.

[15] Křoustek, J.; Zemek, P.: *Kategorizace malware vzorků*. Interná dokumentácia spoločnosti AVG Technologies.

[16] Liu, K.; Tan, H. B. K.; Chen, X.: *Binary code analysis*. *Computer [online]*, ročník 46, č. 8, 2013, ISSN 0018-9162.

[17] Oracle: *JAR File Overview*. [Online; cit. 17. 01. 2017].
URL docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html

[18] Panakkal, G. R.: *Leaving our ZIP undone: How to abuse ZIP to deliver malware apps*. In *Virus Bulletin Conference*, 2014, [Online; cit. 17. 01. 2017].
URL
www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Panakkal.pdf

[19] Peijnenburg, F.: *Security in Android apps*. 2013, [Online; cit. 20. 01. 2017].
URL www.cs.uu.nl/docs/vakken/b3sec/Proj12/Android.pdf

[20] PKWARE, I.: *APPNOTE.TXT – .ZIP File Format Specification*. 2012, [Online; cit. 10. 01. 2017].
URL pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.3.TXT

[21] PyPy: *Welcome to PyPy*. [Online; cit. 08.04.2017].
URL pypy.org/features.html

[22] Rokach, L.; Maimon, O.: *Data Mining and Knowledge Discovery Handbook*. Springer, 2005, ISBN 978-0-387-25465-8.

[23] Roundy, K. A.: *Hybrid analysis and control of malicious code*. Dizertační práce, Univeristy of Wisconsis-Madison, 2012.
URL pages.cs.wisc.edu/~roundy/dissertation.pdf

[24] Seguin, K.: *The Little MongoDB Book*. 2011, [Online; cit. 10. 01. 2017].
URL openmymind.net/mongodb.pdf

[25] Singh, N.; Khurmi, S. S.: *Malware Analysis, Clustering and Classification: A Literature Review*. In *International Journal of Computer Science and Technology*, *1*, ročník 6, 2015, ISSN 0976-8491.

[26] Slavotínek, T.: *Knihovna pro práci s videem pro platformu Android*. Bakalářská práce, Fakulta informačních technológií, Vysoké Učení Technické v Brně, 2014.

[27] Tan, P.-N.; Kumar, V.; Steinbach, M.: *Introduction to data mining*. Boston: Pearson Addison Wesley, 2006, ISBN 0-321-32136-7, 769 s.

[28] Wikipedia: *DBSCAN*. [Online; cit. 20. 01. 2017].
URL commons.wikimedia.org/wiki/File:DBSCAN-density-data.svg

[29] Wißfeld, M.: *ArtHook – Callee-side method hook injection on the new Android runtime ART*. Bakalářská práce, Saarland University, Faculty of Natural Sciences and Technology, 2015.

[30] Zaki, M.; Meira, W.: *Data mining and analysis*. Cambridge university press, 2014, ISBN 978-0-521-76633-3.

[31] Zendulka, J.; Bartík, V.; Lukáš, R.; aj.: *Získávání znalostí z databází – studijní opora*. 2010, [cit. 19. 01. 2017].

[32] Zhou, W.; Zhou, Y.; Jiang, X.; aj.: *Detecting repackaged smartphone applications in third-party android marketplaces*. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, ACM, 2012, ISBN 978-1-4503-1091-8, s. 317 – 326.

Prílohy

Príloha A

Log zo spustenia zhlukovej analýzy

```
INFO: starting clustering of samples in 'apk-samples/'
INFO: starting analysis of 22713 samples in 'apk-samples/'
INFO: analyzed 22713 samples from 'apk-samples/' in 38m 15s
INFO: starting clustering of samples in 'apk-samples/'
INFO: ==== starting clustering of 'apk' samples in 'apk-samples/' ====
INFO: 'apk': loaded 22713 samples in 67.255s
INFO: 'apk': using 'DEX methods per class (100% match)' to cluster 22707/22713
    samples (min cluster size: 50)
INFO: 'apk': created 35 clusters with 9777 samples via 'DEX methods per class (100%
    match)' in 7.216s (12936 samples remaining)
INFO: 'apk': using 'DEX classes (similarity)' to cluster 12828/12936 samples (min
    cluster size: 50)
INFO: 'apk': created 10 clusters with 671 samples via 'DEX classes (similarity)' in
    7.725s (12265 samples remaining)
INFO: 'apk': using 'Archive member sizes (100% match)' to cluster 12265/12265
    samples (min cluster size: 50)
INFO: 'apk': no clusters via 'Archive member sizes (100% match)' in 1.289s
INFO: 'apk': using 'Archive member sizes (similarity)' to cluster 12265/12265
    samples (min cluster size: 50)
INFO: 'apk': created 4 clusters with 262 samples via 'Archive member sizes (
    similarity)' in 3.955s (12003 samples remaining)
INFO: 'apk': using 'Archive members (100% match)' to cluster 12003/12003 samples (
    min cluster size: 50)
INFO: 'apk': created 1 cluster with 66 samples via 'Archive members (100% match)' in
     1.332s (11937 samples remaining)
INFO: 'apk': using 'Archive members (similarity)' to cluster 11937/11937 samples (
    min cluster size: 50)
INFO: 'apk': created 5 clusters with 383 samples via 'Archive members (similarity)'
    in 5.277s (11554 samples remaining)
INFO: 'apk': using 'APK Chadron features (similarity)' to cluster 5483/11554 samples
     (min cluster size: 50)
INFO: 'apk': created 1 cluster with 57 samples via 'APK Chadron features (similarity
    )' in 0.279s (11497 samples remaining)
...
INFO: 'apk': clustering finished; created 873 clusters with 21676 samples in 39.561s
     (1037 samples unclustered)
INFO: finished clustering of 22713 samples in 'apk-samples/' in 107.600s
INFO: created 874 clusters in 2 categories containing 22713 samples in 1m 51s
INFO: computing detection statuses for clusters in the result
INFO: finished computation of detection statuses in 0.37s
INFO: adding the result into our clustering database
INFO: clustering finished
```

Kód A.1: Log zo zhlukovej analýzy 22 713 APK vzoriek

Príloha B

Ukážka APK zhlukov

[-] **Cluster 1/873 (1940 samples)** by DEX methods per class (100% match)	
APK permissions sim:	100% (all 33 permissions in common)
Archive member sizes sim:	84% (21 sizes in common)
Archive members sim:	92% (26 members in common)
Average ssdeep sim:	57%
DEX API classes sim:	100% (all 90 classes in common)
DEX IP addresses sim:	100% (all 3 addresses in common)
DEX URL addresses sim:	100% (all 9 addresses in common)
DEX classes sim:	80% (356 classes in common)
DEX methods sim:	91% (2189 methods in common)
Detections (VT min):	25/49 (0 unknown)
Detections (VT):	BitDefender: Android.Adware.Dowgin.DX, ESET-NOD32: a variant of idOS.Inoco.a, Qihoo-360: Trojan.Android.Gen
Summary (Chadron):	Malware: 110, Unknown: 1830
Summary (CyberCapture):	Unknown: 1940
Summary (Scavenger):	Pup: 1562, Unknown: 378
Summary (virdat):	infected: 1940
[-] **Cluster 17/873 (149 samples)** by APK main activities (100% match)	
APK main activities:	com.mobimento.caponate.MainActivity
APK permissions sim:	57% (6 permissions in common)
Archive member sizes sim:	22% (64 sizes in common)
Archive members sim:	25% (84 members in common)
DEX API classes sim:	51% (156 classes in common)
DEX URL addresses sim:	15% (10 addresses in common)
Detections (VT min):	4/53 (0 unknown)
Summary (Chadron):	Clean: 74, Unknown: 75
Summary (CyberCapture):	Unknown: 149
Summary (Scavenger):	Malware: 42, Suspicious: 107
Summary (virdat):	infected: 122, unknown: 27
[-] **Cluster 47/873 (76 samples)** by DEX classes (similarity)	
DEX API classes sim:	100% (all 6 classes in common)
DEX classes sim:	100% (all 2 classes in common)
DEX methods sim:	35% (11 methods in common)
Detections (VT min):	1/55 (0 unknown)
Summary (Chadron):	Clean: 27, Malware: 2, Suspicious: 18, Unknown: 29
Summary (CyberCapture):	Unknown: 76
Summary (Scavenger):	Malware: 34, Pup: 16, Suspicious: 25, Unknown: 1
Summary (virdat):	infected: 47, unknown: 29

Obr. B.1: Ukážka vytvorených APK zhlukov