VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO DETEKCI VZORŮ V BINÁRNÍCH SOUBORECH

SYSTEM FOR PATTERN RECOGNITION IN BINARY FILES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MAREK MILKOVIČ
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETER MATULA
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20063/2016/xmilko01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Milkovič Marek, Bc.**

Obor: Bezpečnost informačních technologií

Téma: **Systém pro detekci vzorů v binárních souborech**
System for Pattern Recognition in Binary Files

Kategorie: Bezpečnost

Pokyny:

1. Seznamte se s problematikou analýzy škodlivého kódu. Zaměřte se na metody související s detekcí vzorů v binárních spustitelných souborech.
2. Seznamte se se systémem Clusty pro shlukovou analýzu potenciálně škodlivých souborů společnosti AVG Technologies.
3. Navrhněte takové metody a analýzy, založené na vybraném formalismu z bodu 1), které rozšíří systém Clusty o možnost detekce vzorů a případně i možnost klasifikace na základě takové detekce.
4. Po konzultaci s vedoucím a konzultantem implementujte analýzy navržené v předchozím bodě. Výsledné řešení by pro zadané vstupní soubory mělo automaticky vygenerovat dostatečně unikátní detekční vzor.
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)
- Křoustek, J.: Retargetable Analysis of Machine Code, PhD thesis, Brno, FUT VUT, 2014.
- Popis jazyku YARA [online]. 2016. <http://yara.readthedocs.org>
- Interní dokumentace systému Clusty, AVG Technologies, 2016.

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání a alespoň rozpracovaný čtvrtý bod.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% 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: **Matula Peter, Ing.**, UIFS FIT VUT

Konzultant: Křoustek Jakub, Ing., UIFS FIT VUT

Datum zadání: 1. listopadu 2016

Datum odevzdání: 24. května 2017

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt
Škodlivý software sa v dobe internetu šíri veľmi rýchlo a poškodzuje užívateľov a ich dáta. Je preto nutné vylepšovať metódy akými pristupujeme k jeho analýze, aby bolo možné chrániť potenciálne obete. Táto práca sa zaoberá návrhom a implementáciou systému na tvorbu detekčných vzorov zo spustiteľných súborov v spolupráci so spoločnosťou AVG Technologies. Cieľom tejto práce je zostrojiť nástroj na generovanie detekčného vzoru z množiny binárnych súborov. V rámci práce sú tiež navrhnuté nové druhy analýz na extrakciu informácií zo spustiteľných súborov. Navrhnutý a implementovaný systém sa používa v praxi pri analýze nového škodlivého kódu a je integrovaný do systému na zhlukovú analýzu.

Abstract
Malicious software spreads really fast in the age of the Internet and it harms users and their data. Therefore, it is necessary to improve methods of how we deal with its analysis, so we can protect potential victims. This thesis deals with design and implementation of system for generating patterns out of executable files in cooperation with AVG Technologies. The goal of this work is to create a tool that generates a detection pattern from the set of binary files. This work further proposes new types of analyses for extraction of information out of executable files. Designed and implemented system is used in practice for analysis of new malicious code and it is integrated into the clustering system.

Kľúčové slová
reverzné inžinierstvo, detekcia vzorov, malware, YARA, AVG

Keywords
reverse engineering, pattern recognition, malware, YARA, AVG

Citácia
MILKOVIČ, Marek. *Systém pro detekci vzorů v binárních souborech.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Matula Peter.

Systém pro detekci vzorů v binárních souborech

Prehlásenie

Prehlasujem, že som túto prácu vypracoval sám pod vedením Ing. Petra Matulu. Ďalšie informácie mi poskytli Ing. Jakub Křoustek, Ph.D., a Ing. Petr Zemek, Ph.D. Uviedol som všetky literárne zdroje a publikácie, z ktorých som čerpal.

. . . .
Marek Milkovič
24. mája 2017

Poďakovanie

Chcel by som poďakovať svojmu vedúcemu Ing. Petrovi Matulovi za poskytnutú odbornú pomoc pri vypracovaní tejto práce. Taktiež by som chcel poďakovať Ing. Jakubovi Křoustkovi, Ph.D., a Ing. Petru Zemkovi, Ph.D. za rady, ktoré mi poskytli. V neposlednej rade chcem tiež poďakovať svojej rodine, ktorá mi dodala podporu pri dokončovaní tejto práce.

Obsah

1 Úvod **3**

2 Formát spustiteľných súborov PE **5**
- 2.1 Sekcie . . . 6
- 2.2 Tabuľka importovaných symbolov . . . 7
- 2.3 Tabuľka certifikátov . . . 7
- 2.4 Tabuľka ladiacich informácií . . . 10
- 2.5 .NET informácie . . . 10
- 2.6 Rich hlavička . . . 13

3 Analýza škodlivého kódu **14**
- 3.1 Statická analýza . . . 14
  - 3.1.1 Hlavičky formátu spustiteľného súboru . . . 15
  - 3.1.2 Importované knižnice a symboly . . . 15
  - 3.1.3 Reťazce . . . 15
- 3.2 Dynamická analýza . . . 16
  - 3.2.1 Prístup k súborovému systému . . . 16
  - 3.2.2 Sieťová komunikácia . . . 17
  - 3.2.3 Registre . . . 17
  - 3.2.4 Pomenované objekty . . . 18

4 Nástroje na analýzu v spoločnosti AVG Technologies **19**
- 4.1 Clusty . . . 19
- 4.2 Fileinfo . . . 21
- 4.3 Cuckoo . . . 23
- 4.4 YARA . . . 24

5 Návrh nových analýz a generovania detekčného vzoru **27**
- 5.1 Nové analýzy . . . 27
  - 5.1.1 Certifikáty a podpis . . . 27
  - 5.1.2 Rekonštrukcia .NET typov . . . 28
  - 5.1.3 Detekcia reťazcov . . . 31
- 5.2 Generovanie detekčného vzoru . . . 32
  - 5.2.1 Extraktor . . . 33
  - 5.2.2 Analyzátor . . . 33
  - 5.2.3 Generátor . . . 35
  - 5.2.4 Konfigurácia . . . 41

6 Implementácia nových analýz a generovania detekčného vzoru 43
- 6.1 Implementácia nových analýz . . . 43
  - 6.1.1 Certifikáty a podpis . . . 43
  - 6.1.2 Rekonštrukcia .NET typov . . . 45
  - 6.1.3 Detekcia reťazcov . . . 47
- 6.2 Generovanie detekčného vzoru . . . 48
  - 6.2.1 Extraktor . . . 48
  - 6.2.2 Analyzátor . . . 49
  - 6.2.3 Generátor . . . 51
  - 6.2.4 Konfigurácia . . . 52
- 6.3 Zdrojové kódy, metriky, kompilácia a spúšťanie . . . 52
  - 6.3.1 Zdrojové kódy . . . 53
  - 6.3.2 Metriky . . . 53
  - 6.3.3 Kompilácia . . . 54
  - 6.3.4 Spustenie . . . 54
- 6.4 Optimalizácie . . . 55

7 Testovanie a výsledky 56
- 7.1 Testovanie generátoru . . . 56
  - 7.1.1 Jednotkové testy . . . 56
  - 7.1.2 Integračné testy . . . 57
  - 7.1.3 Regresné testy . . . 58
- 7.2 Vyhodnotenie kvality detekčných vzorov . . . 59
- 7.3 Integrácia a nasadenie v praxi . . . 60
  - 7.3.1 Clusty . . . 60
  - 7.3.2 Ransomware WannaCry . . . 61

8 Záver 63

Literatúra 65

Kapitola 1

Úvod

Ľudská spoločnosť sa aktuálne nachádza v informačnej dobe. S tým spojený rozvoj informačných a komunikačných technológií nás ovplyvnil v každom aspekte nášho života. Elektronické bankovníctvo, informatizácia verejnej správy, sociálnej siete, to sú všetko technológie, ktoré sa dnes dotýkajú už takmer polovice populácie, pretože podľa štatistík je v roku 2016 pripojených k internetu približne 3.5 miliardy ľudí [27]. O mnohých z nás sa v elektronickom svete nachádza čoraz viac a viac informácií. Tak isto ako v reálnom svete existujú ľudia ochotní porušiť zákon za vidinou osobného obohatenia, tak podobní ľudia existujú aj v tom svete virtuálnom.

V tejto súvislosti sa vo svete softwaru môžeme hovoriť o tzv. *malware* (angl. *malicious software* – škodlivý software), ktorého účelom je práve ukradnúť, či poškodiť dáta užívateľa. *Malware* pritom zastrešuje veľa rôznych typov škodlivého kódu, ako sú víry, červy, trójske kone, *ransomware*[1] a iné. Podľa štatistiky spoločnosti Kaspersky Lab bolo v treťom kvartáli roku 2016 registrovaných vyše 12 miliónov výskytov nového *malwaru* [24]. Užívateľom však často chýba povedomie o tom ako sa chrániť a aké sú zásady bezpečnosti. Je preto nutné sa *malwarom* zaoberať a študovať ho, aby sme boli schopný vyvíjať a zlepšovať mechanizmy na jeho detekciu a poskytovať užívateľom prostriedky, ktoré ich chránia.

Z toho dôvodu vykonávame analýzu škodlivého kódu. Pod týmto pojmom sa rozumie proces porozumenia toho čo daný škodlivý kód vykonáva, aké sú následky jeho spustenia a ako ho eliminovať. V rámci spoločnosti AVG Technologies sa k zlepšeniu analýzy škodlivého kódu vyvinul systém pre zhlukovú analýzu *malwaru* nazývaný Clusty. Jeho úlohou je pomôcť analytikom rozdelením nových spustiteľných súborov do zhlukov na základe spoločných vlastností.

Motiváciou tejto práce je boj proti novému *malwaru* a experimentovanie s možnosťami ako ho odhalovať a pritom zrýchliť a zefektívniť proces analýzy. Táto práca si kladie dva hlavné ciele. Tým prvým je vylepšenie analýz, ktoré systém Clusty už vykonáva. Druhým cieľom tejto práce je vytvoriť nástroj, ktorý by automaticky vygeneroval detekčný vzor zhluku spustiteľných súborov vo formáte YARA. Tento nástroj by mohol byť používaný ako z Clustyho pre tvorbu vzorov z vytvorených zhlukov, tak aj samostatne pre vlastnú analýzu.

Aj napriek tomu, že sa *malware* netýka len platformy Windows, tak podľa bezpečnostnej správy za rok 2015/2016 od nezávislého bezpečnostného inštitútu AV-Test je práve hlavným cieľom útokov platforma Windows, pričom až 99.69% z *malwaru* pre túto platformu tvoria

[1] *Malware*, ktorý drží súbory či zariadenie užívateľa za účelom vydierania a požadovania výkupného.

32-bitové programy [19]. Z toho dôvodu sa bude táto práca hlavne zaoberať platformou Windows.

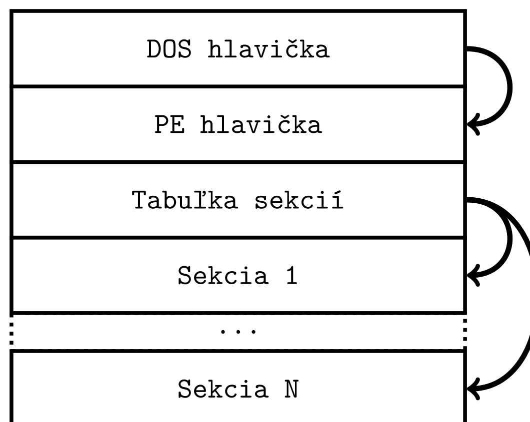
Práca je členená na kapitoly, ktoré obsahujú podkapitoly pre lepší prehľad. Obsah tejto práce začína popisom formátu spustiteľných súborov použitých na platforme Windows v kapitole 2. Kapitola 3 sa venuje všeobecnej analýze *malwaru* pre formát súborov popísaných v prvej kapitole. Kapitola 4 načŕta aké nástroje sa používajú v spoločnosti AVG Technologies na analýzu. Kapitola 5 navrhuje nové analýzy, ktoré by bolo možné zabudovať do existujúcich nástrojov a taktiež návrh nástroja na generovanie detekčného vzoru. Implementácia podľa návrhu je rozoberaná v rámci kapitoly 6. Testovanie a vyhodnotenie výsledných detekčných vzorov je rozobrané v kapitole 7. Kapitola 8 uzatvára túto prácu zhodnotením dosiahnutých cieľov a pojednáva o budúcom rozvoji navrhnutých riešení.

Kapitola 2

Formát spustiteľných súborov PE

Táto práca sa zaoberá analýzou programov na platforme Windows. Z toho dôvodu je nutné si predstaviť formát, v akom sú tieto súbory uložené, aby sme pochopili, ktoré štruktúry je vhodné analyzovať a využiť pri tvorbe detekčného vzoru. Spomínaný formát sa nazýva PE (*Portable Executable*). Kapitola čerpá informácie hlavne z oficiálnej dokumentácie formátu PE [31] a z mojej vlastnej bakalárskej práce [33].

PE je formát spustiteľných súborov, ktorý je používaný na platforme Windows pre súbory s príponou EXE a DLL. Môžeme sa tiež stretnúť s názvom PE/COFF nakoľko bol vytvorený zo staršieho formátu COFF určeného pre systémy Unix.



Obr. 2.1: Štruktúra spustitelného súboru v PE formáte [33].

Základná štruktúra súboru vo formáte PE je znázornená na obrázku 2.1. Na začiatku súboru sa nachádza DOS hlavička, niekedy označovaná aj ako MZ hlavička podľa úvodných dvoch bajtov `MZ` podľa iniciálok tvorcu formátu. Jedná sa o rovnakú hlavičku, ktorá sa používala v starých DOS programoch. To robí súbory vo formáte PE spustiteľné aj na historickom DOSe, avšak táto hlavička zvyčajne odkazuje len na program, ktorý vypíše hlásenie `This program can not be run in DOS mode`.

Za DOS hlavičkou sa nachádza PE hlavička, obsahujúca informácie dôležité pre načítanie a spustenie tohto programu. Medzi tieto informácie patria hodnoty ako adresa vstupného bodu programu, veľkosť zásobníku a haldy, počet sekcií, či informácie o tzv. *data directories*.

Data directory je dátová štruktúra definujúca dodatočné informácie o programe. Hlavičky všetkých *data directories* sa skladajú len z 2 hodnôt a to z adresy a veľkosti daného

directory. Existuje 16 rôznych typov *data directories*. Medzi tie najzaujímavejšie a podstatné pre túto prácu patria nasledovné.

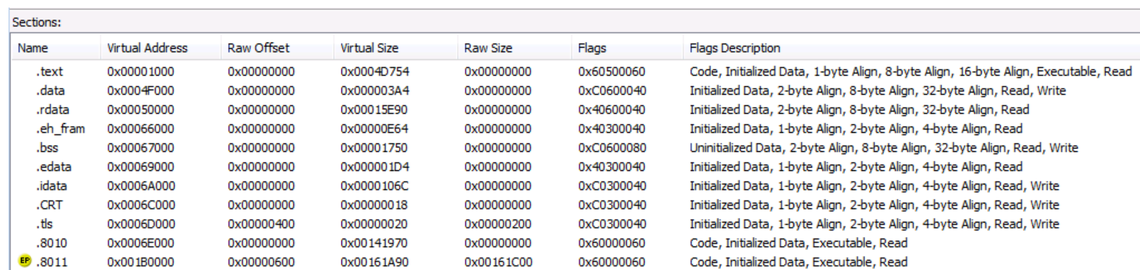
- *Import directory* obsahujúci informácie o importovaných symboloch, ako sú funkcie, či premenné. Ďalej v práci môže byť táto štruktúra označovaná aj ako tabuľka importovaných symbolov.
- *Certificate/security directory* obsahujúci informácie o certifikáte autora aplikácie a o digitálnom podpise obsahu súboru. Ďalej v práci môže byť táto štruktúra označovaná aj ako tabuľka importovaných symbolov.
- *CLR directory* obsahujúci informácie o .NET hlavičke. Programy pre platformu .NET framework sa totiž nachádzajú v PE obálke.
- *Debug directory* obsahujúci informácie pre ladenie programu ako sú názvy symbolov, asociácie inštrukcií a riadkov kódu v programe a iné. Ďalej v práci môže byť táto štruktúra označovaná aj ako tabuľka ladiacich informácií.

Posledné dve nepatria medzi typické *data directories*, ktoré sa nachádzajú v každom spustiteľnom súbore, avšak budeme ich potrebovať neskôr v tejto práci. Preto si ich spolu s tými typickými popíšeme podrobnejšie.

2.1 Sekcie

Sekcie delia obsah programu na logické celky. Ten sa totiž neskladá len z jedného druhú dát, ale obsahuje kód, konštantné dáta, variabilné dáta, či dáta s rôznou inou sémantikou. Pre každý takýto druh dát vytvorí prekladač vlastnú sekciu. Tým môže pomôcť operačnému systému s tým so správnym nastavením prístupových práv k stránkam pamäti.

Každá sekcia ma priradené meno, ktoré nemusí byť unikátne. Každý prekladač však zvykne pomenovávať sekcie podľa ich obsahu zvyčajne rovnakým menom. Napríklad `.text` je zvyčajný názov sekcie obsahujúcej kód, `.rdata` pre konštantné dáta, `.data` pre variabilné dáta a `.bss` pre neinicializované dáta. Ďalej má každá sekcia priradenú fyzickú adresu a fyzickú veľkosť. Tie vymedzujú, kde v súbore sa nachádza obsah sekcie. Popri fyzickej majú sekcie priradenú aj logickú adresu a veľkosť, ktoré zase vymedzujú umiestnenie sekcie v rámci adresného priestoru procesu. Ukážka sekcií v *malwari* Adylkuzz, určeného na ťaženie kryptomeny, je na obrázku 2.2.



Sections:

Name	Virtual Address	Raw Offset	Virtual Size	Raw Size	Flags	Flags Description
.text	0x00001000	0x00000000	0x0004D754	0x00000000	0x60500060	Code, Initialized Data, 1-byte Align, 8-byte Align, 16-byte Align, Executable, Read
.data	0x0004F000	0x00000000	0x000003A4	0x00000000	0xC0600040	Initialized Data, 2-byte Align, 8-byte Align, 32-byte Align, Read, Write
.rdata	0x00050000	0x00000000	0x00015E90	0x00000000	0x40600040	Initialized Data, 2-byte Align, 8-byte Align, 32-byte Align, Read
.eh_fram	0x00066000	0x00000000	0x00000E64	0x00000000	0x40300040	Initialized Data, 1-byte Align, 2-byte Align, 4-byte Align, Read
.bss	0x00067000	0x00000000	0x00001750	0x00000000	0xC0600080	Uninitialized Data, 2-byte Align, 8-byte Align, 32-byte Align, Read, Write
.edata	0x00069000	0x00000000	0x000001D4	0x00000000	0x40300040	Initialized Data, 1-byte Align, 2-byte Align, 4-byte Align, Read
.idata	0x0006A000	0x00000000	0x0000106C	0x00000000	0xC0300040	Initialized Data, 1-byte Align, 2-byte Align, 4-byte Align, Read, Write
.CRT	0x0006C000	0x00000000	0x00000018	0x00000000	0xC0300040	Initialized Data, 1-byte Align, 2-byte Align, 4-byte Align, Read, Write
.tls	0x0006D000	0x00000400	0x00000020	0x00000200	0xC0300040	Initialized Data, 1-byte Align, 2-byte Align, 4-byte Align, Read, Write
.8010	0x0006E000	0x00000000	0x00141970	0x00000000	0x60000060	Code, Initialized Data, Executable, Read
EP .8011	0x001B0000	0x00000600	0x00161A90	0x00161C00	0x60000060	Code, Initialized Data, Executable, Read

Obr. 2.2: Ukážka sekcií v *malwari* Adylkuzz.

2.2 Tabuľka importovaných symbolov

V programoch nevyužívame vždy len kód, ktorý si sami napíšeme. Skoro vždy využívame knižnice, či už systémové alebo užívateľské, obsahujúce cudzí kód. Tento *data directory* definuje, ktoré symboly chceme využiť práve z týchto knižníc. Pri načítavaní programu do pamäte sa potom operačný systém postará, aby symboly spomenuté v tejto dátovej štruktúre boli zavedené do adresného priestoru nášho procesu.

Import data directory môžeme rozdeliť na dve časti a to ILT (*Import Lookup Table*) a IAT (*Import Address Table*). ILT obsahuje zoznam symbolov ktoré je nutné pri načítaní programu importovať. Ten môže byť buď vo forme názvov, ale aj priamo indexom do tabuľky exportov danej knižnice, tzv. import ordinálom. IAT obsahuje adresy jednotlivých importovaných funckií. Spočiatku IAT neobsahuje žiadne adresy, ale pri prechádzaní ILT je naplnená správnymi adresami. Počas svojho behu používa program adresy z IAT.

Praktická ukážka niektorých zaujímavých importovaných symbolov v *ransomwari* Spora je možné vidieť na obrázku 2.3.

Import: 10 modules, 0 missed

Module	Delayed
ADVAPI32.dll	
CRYPT32.dll	
KERNEL32.dll	
MPR.dll	
ole32.dll	
oleaut32.dll	
SHELL32.dll	
SHLWAPI.dll	
USER32.dll	
VERSION.dll	

20 functions, 0 missed

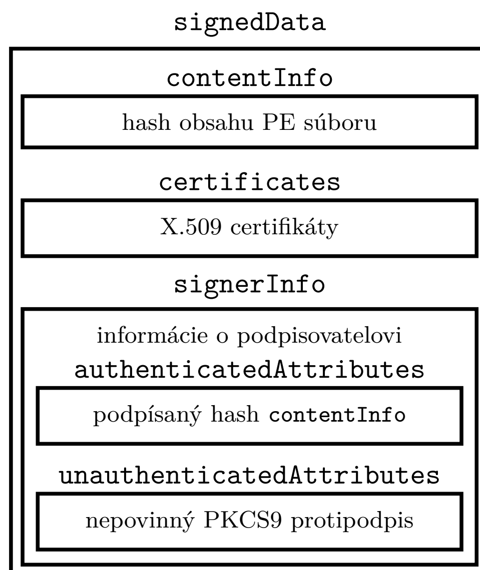
Name	Hint	Module
CryptAcquireContextW	173	ADVAPI32.dll
CryptCreateHash	175	ADVAPI32.dll
CryptDecrypt	176	ADVAPI32.dll
CryptDestroyHash	178	ADVAPI32.dll
CryptDestroyKey	179	ADVAPI32.dll
CryptEncrypt	182	ADVAPI32.dll
CryptExportKey	187	ADVAPI32.dll
CryptGenKey	188	ADVAPI32.dll
CryptGetHashParam	192	ADVAPI32.dll
CryptHashData	196	ADVAPI32.dll
CryptImportKey	198	ADVAPI32.dll
CryptReleaseContext	199	ADVAPI32.dll
GetSidSubAuthority	337	ADVAPI32.dll
GetSidSubAuthorityCount	338	ADVAPI32.dll
GetTokenInformation	340	ADVAPI32.dll
GetUserNameA	350	ADVAPI32.dll
OpenProcessToken	497	ADVAPI32.dll
RegCloseKey	554	ADVAPI32.dll
RegDeleteValueW	578	ADVAPI32.dll
RegOpenKeyExW	603	ADVAPI32.dll

Obr. 2.3: Ukážka importovaných symbolov v *ransomwari* Spora.

2.3 Tabuľka certifikátov

Microsoft dáva možnosť vývojárom podpísať obsah spustiteľného súboru. Do spustiteľného súboru je potom umiestnený certifikát autora aplikácie spolu so zvyškom reťazca certifikátov certifikačných autorít. Táto technológia sa nazýva *Microsoft Authenticode* [32]. To umožňuje si overiť, či sedí podpis obsahu súboru a skutočný obsah súboru, ktorý mohol byť upravený. Správny podpis však nie je záruka bezpečnosti. Vyskytuje sa mnoho podpísaného *malwaru* a tento trend stúpa [15, 11]. Historicky *malware* podpisy nepoužíval, čo ho znevýhodňovalo u antivírusových firiem. To ale vyvolalo reakciu autorov *malwaru*, ktorí začali svoje výtvory podpisovať napríklad odcudzenými certifikátmi inak dôveryhodných aplikácií.

Certifikáty spolu s informáciami o podpise obsahu súboru sú umiestnené práve v tomto *data directory*. Všetky informácie sú uložené v štruktúre založenej na štandarde PKCS7 [28] (angl. *Public Key Cryptography Standards 7* – štandard kryptografie s verejným kľúčom) pričom certifikáty sú vo formáte X.509v3 [21]. Obsah tejto štruktúry je uložený vo forme zakódovaného formátu ASN.1[1] [25] (*Abstract Syntax Notation One*) pomocou kódovania DER [26] (*Distinguished Encoding Rules*). Jedná sa o binárne kódovanie vo formát TLV (*Type-Length-Value* – typ-dĺžka-hodnota), pričom každý typ ASN.1 formátu je zakódovaný jedinečne. Špecifikácia *Microsoft Authenticode* definuje obsah PKCS7, ako je to naznačené na obrázku 2.4.



Obr. 2.4: Štruktúra PKCS7.

Každý certifikát obsahuje niekoľko atribútov, medzi ktoré patria napríklad:

- Subjekt resp. vlastník certifikátu
- Vystavovateľ certifikátu
- Sériové číslo
- Verejný kľúč (algoritmus + samotný kľúč)
- Dátum platnosti (interval od – do)

Subjekt a vystavovateľ sú často prezentovaný formou ich identifikátora, ktorý sa skladá z jednotlivých atribútov samotného subjektu resp. vystavovateľa uložených vo forme RDN (angl. *Relative Distinguished Names* – relatívne jedinečné mená). V tomto formáte ma každý atribút pridelený svoj unikátny kľúč a výsledný identifikátor je potom reťazec skladajúci sa zo sekvencie kľúčov atribútov a ich hodnôt. Medzi atribúty subjektu resp. vystavovateľa patrí napríklad (v zátvorke je uvedený unikátny kľúč atribútu):

- Meno (`CN`)

[1] ASN.1 je abstraktná notácia pre popis abstraktných typov a hodnôt.

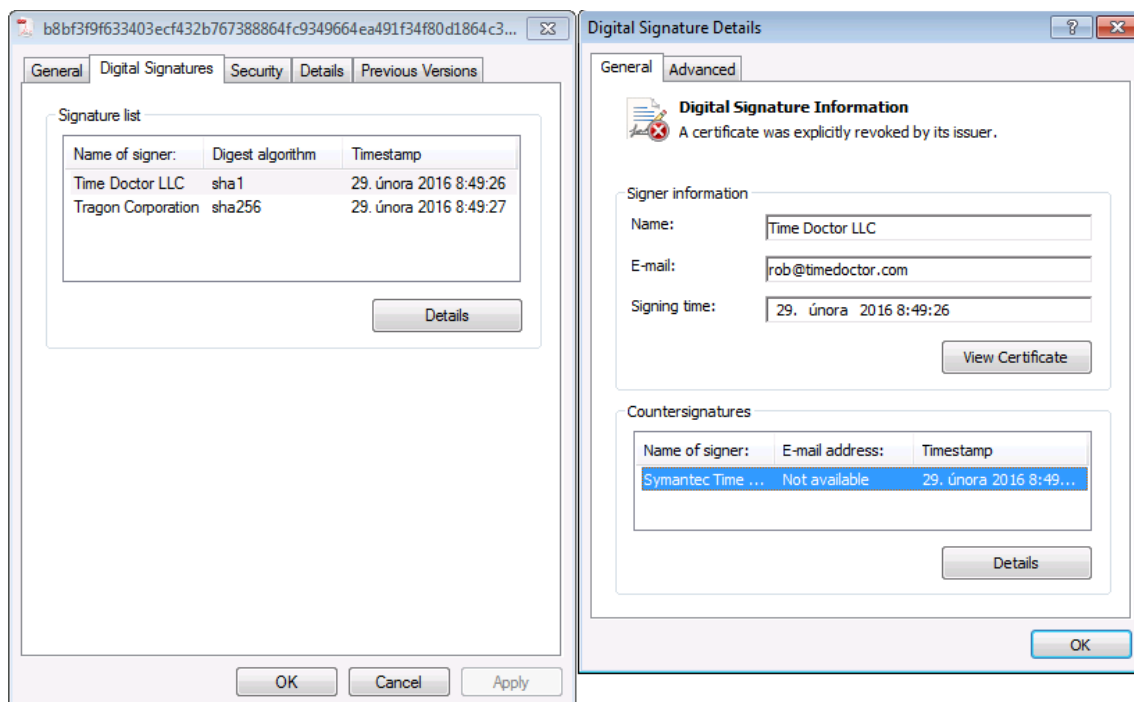
- Názov organizácie (O)
- Krajina (C)
- Lokácia/mesto (L)
- Štát (ST)

Identifikátorom subjektu resp. vystavovateľa je potom reťazec poskladaný z jednotlivých atribútov oddelených oddelovačom. Napríklad C=CZ/L=Brno/CN=Milkovič/O=FIT.

Podpis obsahu súboru je uložený v authenticatedAttributes atribúte v rámci položky signerInfo. Nejedná sa však priamo o podpis samotného obsahu, ale o podpis atribútu contentInfo, v ktorom je uložený hash obsahu súboru. Tento hash sa počíta priamo z celého obsahu, výnimkou sú však konkrétne tieto 3 časti, ktoré sa vynechávajú.

- *Checksum* (kontrolný súčet) v PE hlavičke (4 bajty)
- Metadáta od *security directory* (8 bajtov)
- Obsah *security directory*

Ukážka digitálne podpísaného *ransomwaru* je možné vidieť na obrázku 2.5. Všimnime si, že certifikát bol revokovaný a názov subjekt odkazuje na Time Doctor LLC, čo značí odcudzený certifikát.



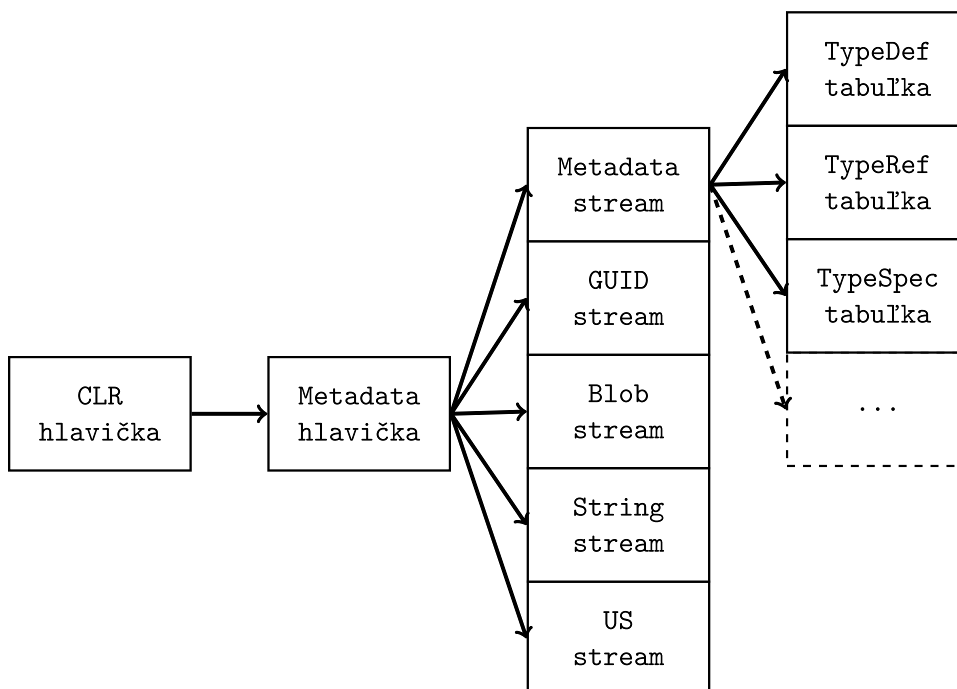
Obr. 2.5: Ukážka podpísaného *ransomwaru* CTBLocker.

2.4 Tabuľka ladiacich informácií

Ladiace informácie sú užitočné hlavne v prípade, že chceme program ladiť počas jeho beho pomocou nejakého ladiaceho nástroja. Obsahujú informácie ako asociácie adries a názvov symbolov, mapovanie inštrukcií na riadky programu a mnoho ďalších. U formátu PE existuje viac typov tabuliek ladiacich informácií, ale najpoužívanejším typom je typ `IMAGE_DEBUG_TYPE_CODEVIEW`. Tento špecifický typ definuje, že ladiaci informácie sú dodané v podobe externého súboru vo formáte PDB (angl. *Program Database*). *Debug directory* v tomto prípade ponúka len plnú cestu k PDB súboru.

2.5 .NET informácie

Pre spustiteľné súbory platformy .NET je použité *CLR directory* na uskladnenie informácií o programe v tomto formáte [22]. Ten je rozdelený do niekoľkých hierarchií. Na prvej úrovni hierarchie sa nachádza tzv. *CLR hlavička* (niekedy sa stretneme s označením *COM hlavička*). Tá obsahuje informácie ako rôzne druhy príznakov ale podstatnou je hlavne adresa a veľkosť štruktúry nazvanej *metadata hlavička*. Tá obsahuje informácie o .NET metadátach a odkazy na tzv. NET *streamy*, v ktorých sa nachádzajú už jednotlivé dáta .NET programu. Typ každého *streamu* určuje o aký druh dát sa jedná, pričom typ je rozlišovaný na základe názvu *streamu*. Celková hierarchia jednotlivých štruktúr je zobrazená na obrázku 2.6.



Obr. 2.6: Hierarchia štruktúr a tabuliek .NET hlavičiek.

Ako je to zobrazená aj na obrázku vyššie, tak existuje dokopy 5 typov *streamov* a to:

- *Metadata stream* (názov `#~` alebo `#-`) – obsahuje tabuľky tried, metód, atribútov a iných typov, ktoré sa nachádzajú v .NET programe.
- *Blob stream* (názov `#Blob`) – obsahuje binárne dáta ako sú napríklad kódovania signatúr metód, typov atribútov a iné.

- *GUID stream* (názov `#GUID`) – obsahuje tabuľku GUID identifikátorov.
- *String stream* (názov `#Strings`) – obsahuje reťazce ako sú názvy tried, metód, parametrov metód, atribútov a iné.
- *User string stream* (názov `#US`) – užívateľské dáta, ktoré nemajú presne danú sémantiku.

Metadata tabuľky

Ako bolo spomenuté, *metadata stream* obsahuje *metadata tabuľky* obsahujúce informácie o dátových typoch. Tieto tabuľky sú uložené sekvenčne za sebou, pričom ich veľkosti sú zapísané sekvenčne priamo pred začiatkom prvej tabuľky. Jednotlivé tabuľky nemajú fixnú veľkosť v počte bajtov. V prípade, že sa atribút záznamu v tabuľke odkazuje do *string streamu*, tak veľkosť indexu sú 2 bajty ak je *string stream* menší ako 2^{16} bajtov, inak má index veľkosť 4 bajty. To isté platí o indexe do ostatných *streamov*, či odkazu do inej tabuľky. V prípade, že sa atribút môže odkazovať do niekoľko tabuliek, tak je nutné odpočítať počet bitov potrebných na označenie o ktorú tabuľku sa jedná. Napríklad ak sa môže index odkazovať do n tabuliek, tak je potrebných $\lceil \log_2(n) \rceil$ bitov na adresovanie. Hranica medzi 2 a 4 bajtami na index je potom $2^{16-\lceil \log_2(n) \rceil}$.

Z dôvodu zachovania stručnosti tejto kapitoly nebudú uvedené atribúty jednotlivých tabuliek, ak napriek tomu že sa ďalej v texte budeme na niektoré z nich odkazovať. Jednotlivé atribúty tabuliek a ich veľkosti je možné nájsť v [22] v prípade potreby a záujmu. Tabuľky v *metadata streame* patriace medzi zaujímavé z pohľadu tejto práce sú:

- `TypeDef` – užívateľsky definované triedy a rozhrania
- `TypeRef` – použité importované triedy a rozhrania z knižníc
- `TypeSpec` – hierarchia dedičnosti tried
- `InterfaceImpl` – hierarchie implementácie rozhraní
- `MethodDef` – metódy definované v užívateľsky definovaných triedach a rozhraniach
- `Field` – atribúty definované v užívateľsky definovaných triedach
- `Property` – vlastnosti užívateľsky definovaných tried
- `PropertyMap` – mapovanie vlastností na konkrétne triedy
- `Param` – parametre užívateľsky definovaných metód
- `NestedClass` – informácie o triedach vnorených v iných triedach
- `GenericParam` – informácie o generických parametroch

Signatúry

Signatúry slúžia ako prostriedok na kódovanie dátových typov atribútov, návratového typu metód, typov formálnych parametrov metód. Sú zapísané v *blob streame*. Kódované sú vo forme regulárneho jazyka nad abecedou elementárnych typov. Elementárny typ je základný

Názov	Hodnota	Popis
ELEMENT_TYPE_VOID	0x01	Typ void.
ELEMENT_TYPE_BOOLEAN	0x02	Typ bool.
ELEMENT_TYPE_CHAR	0x03	Typ char.
ELEMENT_TYPE_I1	0x04	Typ sbyte.
ELEMENT_TYPE_U1	0x05	Typ byte.
ELEMENT_TYPE_I2	0x06	Typ short.
ELEMENT_TYPE_U2	0x07	Typ ushort.
ELEMENT_TYPE_I4	0x08	Typ int.
ELEMENT_TYPE_U4	0x09	Typ uint.
ELEMENT_TYPE_I8	0x0A	Typ long.
ELEMENT_TYPE_U8	0x0B	Typ ulong.
ELEMENT_TYPE_R4	0x0C	Typ float.
ELEMENT_TYPE_R8	0x0D	Typ double.
ELEMENT_TYPE_STRING	0x0E	Typ string.
ELEMENT_TYPE_PTR	0x0F	Ukazovateľ. Nasleduje signatúra ďalšieho typu.
ELEMENT_TYPE_BYREF	0x10	Referencia ref. Nasleduje signatúra ďalšieho typu.
ELEMENT_TYPE_VALUETYPE	0x11	Odkaz na triedu. Nasleduje odkaz do TypeDef alebo TypeRef.
ELEMENT_TYPE_CLASS	0x12	Odkaz na triedu. Nasleduje odkaz do TypeDef alebo TypeRef.
ELEMENT_TYPE_VAR	0x13	Generický parameter u triedy. Nasleduje index parametru pre daný typ.
ELEMENT_TYPE_ARRAY	0x14	Pole. Nasleduje typ prvkov, počet dimenzií a ich hranice.
ELEMENT_TYPE_TYPEDBYREF	0x16	Typ TypedReference.
ELEMENT_TYPE_INTPTR	0x18	Typ IntPtr.
ELEMENT_TYPE_UINTPTR	0x19	Typ UIntPtr.
ELEMENT_TYPE_FNPTR	0x1B	Ukazovateľ na funkciu.
ELEMENT_TYPE_OBJECT	0x1C	Typ object.
ELEMENT_TYPE_SZARRAY	0x1D	Jednorozmerné pole s dolným rozmerom 0. Nasleduje signatúra ďalšieho typu.
ELEMENT_TYPE_MVAR	0x1E	Generický parameter u metódy. Nasleduje index parametru pre daný typ.
ELEMENT_TYPE_CMOD_REQD	0x1F	Špeciálny modifikátor typu. Nasleduje odkaz do TypeDef alebo TypeRef.

Tabuľka 2.1: Elementárne typy.

stavebný prvok komplexných dátových typov. Existuje mnoho elementárnych typov, niektoré sú jednoduché, iné zase vyžadujú prítomnosť ďalších elementárnych typov, či indexov do niektorej z *metadata* tabuliek. Tie zaujímavé z pohľadu tejto práce sú na tabuľke 2.1.

Pre elementárny typ `ELEMENT_TYPE_CMOD_REQD` platí, že modifikátory môžu byť rôzny typ v závislosti od prekladača. Napríklad kľúčové slovo `volatile` je riešené ako modifikátor, ktorý sa odkazuje na typ `System.Runtime.CompilerServices.IsVolatile`.

TypeLib identifikátor

Takmer každý .NET program obsahuje túto hodnotu. Jedná sa o 256 bitový generický unikátny identifikátor (GUID). Aj napriek tomu, že sa jedná o GUID, zapísaný býva v *string streame*. Je špecifický tým, že žiadne 2 programy by nemali mať rovnaký *TypeLib* identifikátor, takže je to pomerne vhodný kandidát na identifikáciu konkrétneho programu aj po tom, čo v ňom boli vykonané zmeny [39]. Zapisujeme ho vo formáte daným regulárnym výrazom `[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}`. Obrázok 2.7 ukazuje TypeLib identifikátor v trójskom koni Immirat.

```
..........€%..€%..................€•....................................€™.
....... ... .................€•......... ....€−....€±. ..... ...€µ..........
.........D....€Á.€Á....D....D. ....Y..KMicrosoft.VisualStudio.Editors.Setti
ngsDesigner.SettingsSingleFileGenerator.11.0.0.0.....T...L...P............
.....€™.............................D.......)..$70ade130-9989-47c7-9fe0-62
3edd6b2db4..'.."..................................... ...€Í.............2.2
1.57.2541...............Copyright Â©  2010-2016.......T..WrapNonExceptionTh
rows..÷...........÷... ...................... ÷................_CorExeMain.
mscoree.dll.....˙%. @.......................................................
```

Obr. 2.7: Ukážka TypeLib identifikátora v trójskom koni Immirat.

2.6 Rich hlavička

Medzi DOS a PE hlavičkou sa hlavne u Microsoft prekladačov vyskytuje štruktúra nazývajúca sa *rich* hlavička (angl. *rich header*). K tejto štruktúre neexistuje oficiálna dokumentácia a jej názov bol odvodený neoficiálne z posledných 4 bajtov tejto hlavičky, ktoré majú hodnotu ASCII znakov `Rich` [34]. Prekladače do tejto hlavičky umiestnia informácie o tom aká verzia prekladača bola použitá pri preklade aplikácie spolu s tým aké verzie knižníc boli použité pri *linkovaní* programu. V súbore sa táto hlavička nachádza v zašifrovanej podobe, avšak je šifrovaná veľmi jednoduchým spôsobom, variantou Vigenèrovej šifry [37] s 4 bajtovým kľúčom a operáciou XOR. Kľúč je zapísaný na konci hlavičky. Obrázku 2.8 zobrazuje podobu zašifrovanej *rich* hlavičky v trójskom koni QakBot.

```
MZ..........˙˙..¸.......@.................................¸.....ş..´.Í!¸.
LÍ!This program cannot be run in DOS mode....$.......E.śŮ.oň..oň..oň..oó..o
ň.cpá..oň..oň..oň.épö..oň.Rich.oň.PE..L...H|HW........ŕ........°...€......#
........Ŕ....@..........................@...................................
.............0..P..................................Ŕ.....................
......................... 2..Đ............................text...Ç˘.......°
................... ..`.rdata...Q...Ŕ...`...Ŕ..............@..@.data........
 ....... ..............@..Ŕ.idata..ń....0.......0..............@..Ŕ........
```

Obr. 2.8: Ukážka zašifrovanej *rich* hlavičky v trójskom koni QakBot.

Kapitola 3

Analýza škodlivého kódu

Existuje mnoho spôsobov akým môžeme pristúpiť k analýze škodlivého kódu, ale podľa toho aké údaje sú pre nás zaujímavé a aké sledujeme rozdeľujeme takúto analýzu na dva základné typy, a to statická a dynamická analýza [36]. Cieľ oboch analýz je rovnaký, zistiť čo najviac informácií o analyzovanom súbore. V nasledujúcich riadkoch popíšeme aký je rozdiel medzi týmito dvoma druhmi analýz a aké vlastnosti sa pri nich najčastejšie skúmajú. Táto kapitola čerpá informácie primárne z [36].

3.1 Statická analýza

Statická analýza sa zaoberá analýzou kódu a dát v súbore bez spúšťania samotného programu. Na takýto druh analýz používame buďto nástroje, ktoré dokážu zobraziť informácie o danom formáte spustiteľného súboru, prípadne nástroje nazývané *disassemblery*, ktoré transformujú inštrukcie v binárnej podobe do jazyku symbolických inštrukcií. To ale vyžaduje rozumieť tomuto kódu pre danú architektúru. *Disassembler* však musí oddeliť kód od dát, čo je možné redukovať na problém zastavenia Turingového stroja [40] a tým pádom sa jedná o nerozhodnuteľný problém. Všeobecne existujú dva heuristické prístupy a to *linear sweep* (lineárny priechod) a *recursive traversal* (rekurzívny priechod) [23]. *Linear sweep* len lineárne prechádza dáta v sekciách označených ako kód a dekóduje inštrukcie sekvenčne. *Recursive traversal* berie do úvahy sémantiku inštrukcií a v prípade skokových inštrukcií sa rekurzívne zanoruje a dekóduje na miestach, kde sa v kóde skáče.

Existujú ešte aj nástroje nazývané *decompilery* (spätné prekladače). Ich úlohou je rekonštrukcia pôvodného zdrojového kódu z binárnej podoby. Aj napriek tomu, že sa pri preklade mnoho informácií o pôvodnom zdrojovom kóde stratí, tak existujú metódy, ktorými je možné robiť približnú rekonštrukciu na základe rôznych druhov analýz. Jeden z reprezentantov spätných prekladačov je aj rekonfigurovateľný spätný prekladač RetDec [14], ktorý je vyvíjaný v rámci spoločnosti AVG Technologies v spolupráci s Fakultou informačných technológií.

Statická analýza býva najčastejšie prvým druhom analýzy, o ktorú sa analytik pokúsi. Jednak je bezpečná, pretože pokiaľ nespustíme daný kód, tak nám žiadne nebezpečenstvo nehrozí. Taktiež nám dokáže pripraviť zázemie pre dynamickú analýzu, pretože nám vie prezradiť na čo presne sa sústrediť, či na sieťovú komunikáciu, prístup na súborový systém, alebo niečo iné. Je tiež veľmi jednoducho vykonateľná, pretože nástroje na statickú analýzu bývajú väčšinou voľne dostupné.

Nevýhodou však je, že statický pohľad na spustiteľný súbor dokáže poskytnúť len obmedzené množstvo informácií, ktoré je možné skrývať napríklad pomocou tzv. *packerov*. Jedná sa o nástroje, ktoré buďto kompresiou resp. šifrovaním skryjú pôvodný obsah spustiteľného súboru; a umiestnia doňho kód, ktorý pri spustení tento pôvodný obsah rozbalí resp. rozšifruje. Nástrojom na dekompresiu takto komprimovaných spustiteľných súborov sa zaoberala moja bakalárska práca [33], kde sú tieto metódy a ich odstraňovanie popísané podrobnejšie. Ďalším spôsobom ako zmiasť nástroje na statickú analýzu sú napríklad obfuskácia kódu, teda zámerná transformácia kódu, tak aby bolo ťažšie pochopiť jeho činnosť, napríklad pomocou vkladania zbytočných inštrukcií, alebo transformáciou inštrukcií na ekvivalentné, ale pre človeka neintuitívne postupnosti inštrukcií.

3.1.1 Hlavičky formátu spustiteľného súboru

Jedno z pomerne častých miest kam sa pozeráme pri statickej analýze sú hlavičky spustiteľného súboru. Tieto hlavičky obsahujú cenné metadáta, ktoré používa operačný systém pri načítavaní spustiteľného súboru do pamäte. Medzi zaujímavé atribúty hlavičiek patria

- Adresa vstupného bodu
- Definované symboly
- Importované knižnice a symboly
- Sekcie
  - Počet
  - Názvy
  - Adresa
  - Veľkosť
- Mnoho ďalších špecifických pre konkrétny formát

3.1.2 Importované knižnice a symboly

Importované knižnice a symboly dokážu prezradiť pomerne dosť informácií o tom, čo daný kód bude vykonávať aj bez jeho spustenia. V prípade, že importované symboly nie sú nijak obfuskované a narazíme napríklad na importovaný symbol `InternetReadFile`[1], tak sa daný kód prezradil, že bude pravdepodobne niekedy počas svojho behu sťahovať súbor z internetu.

3.1.3 Reťazce

V rámci súboru je možné hľadať sekvencie tlačiteľných znakov ktoré sú dlhšie ako nejaká prahová hodnota. Aj napriek tomu, že niekoľko z týchto reťazcov nemusí mať žiaden význam, tak sa medzi nimi môžu vyskytovať reťazce, ktoré sú pre analytika cenné ako napríklad URL adresy, cesty k súborom, kľúče registrov a atď.

V reťazcoch je možné nájsť veľmi cenné informácie, často krát také, ktoré by za behu neboli dostupné. Napríklad ukážka z *ransomwaru* NMoreira, ktorý obsahuje odkaz od autorov je na obrázku 3.1.

[1] Jedná sa o funkciu z Win32 API, presnejšie z knižnice `wininet.dll`

```
Hi debugger dude, asm is very cool to understand is not it? I used to crack lots of softwares with Ollyd
bg but lately I don't have time to do it. Good disassembling man or woman lol. Fwosar you are the man, I
 am inspired by dudes who understand what they do. Your bruteforcing tool was amazing, I am really impre
ssed :). I was so stupid to use SHA1, the limited set of characters for the AES password and not setting
 up the IV correctly... Lesson to be learned: This polite idiot here really needs to stop using the pred
icable and insecure srand, RTLGenRandom all the way stupid me. This still idiot person hope you can brea
k this too, I'm not being sarcastic, you're really inspiring. Hugs, NMoreira Core Dev. Btw, can anyone i
n this world guess what NMoreira really means?
```

Obr. 3.1: Ukážka cenného reťazca v *ransomwari* NMoreira.

3.2 Dynamická analýza

Dynamická analýza spočíva v spustení spustiteľného súboru a následne v pozorovaní toho, čo sa deje pri behu programu. Toto pozorovanie môže byť len čisto manuálne, ale môže dochádzať aj k automatizovanému zberu informácií a ich následnému vyhodnoteniu. Na tieto účely sa môžu používať nástroje ako napríklad *debuggery*, ktoré slúžia na krokovanie programov. Je však potrebné skúmaný program spustiť vo virtualizovanom prostredí, hlavne pokiaľ nepoznáme jeho pôvod a máme podozrenie, že sa jedná o *malware*. *Debuggery* však nepostačia na všetko. Buď nemajú žiadne programovateľné rozhranie, alebo je toto rozhranie obmedzené len na nastavovanie *breakpointov* a zobrazovanie hodnôt premenných. Pokiaľ by chceme zoznam všetkých systémových volaní, tak by neposlúžili veľmi dobre.

V tomto prípade sú zaujímavejšie *sandboxy*. Jedná sa o plne kontrolované prostredie v ktorom je možné spúšťať aplikácie. Toto prostredie je buďto emulované, alebo sa využíva tzv. *hookovanie*[2] systémových volaní. Program bežiaci v *sandboxe* by nemal mať žiaden prístup mimo *sandbox* a preto sú ideálne na spúšťanie kódu z neznámeho zdroja.

Oproti statickej analýze, dynamická analýza odhalí pravé správanie programu, pretože napríklad existencia importovaného symbolu, či raťazca neznamená, že sú skutočne aj použité. Môže odhaliť dáta, ktoré boli počas statickej analýzy nedostupné, čí kód, ktorý nebolo možné detekovať pri *disassemblovaní*.

Tak ako u statickej analýzy je možné skrývať dáta a kód a minimálne znepríjemniť túto analýzu, tak aj do behu programu je možné vkladať kontroly na to či je daný program spustený v *debuggeri*, prípadne či sa nachádza v prostredí *sandboxu* a negatívne ovplyvniť výsledok dynamickej analýzy. Je preto nutné aby *debugger*, či *sandbox* skrýval svoju prítomnosť a spustený program tak nemohol rozoznať, či beží na skutočnom systéme, alebo v kontrolovanom prostredí. *Sandboxy* tiež majú nevýhodu toho, že programy sa v nich nezvyknú spúšťať na dlhú dobu a ak by sa aj spúšťali, bolo by to plytvanie prostriedkov. Niekedy je totiž nutné aby bol program spustený dlhšiu dobu, aby sa prejavila jeho skutočná činnosť. Moderné *sandboxy* by preto mali byť schopné simulovať zrýchlenie času a odhaliť aj takéto pokusy o oklamanie kontrolovaného prostredia.

3.2.1 Prístup k súborovému systému

Pri skúmaní správania aplikácie je jedna zo zaujímavých vlastností kam program pristupuje v súborovom systéme. Škodlivý kód sa totiž môže pokúšať siahať do citlivých miest v užívateľských dátach, či systémových konfiguračných súboroch a skúšať, či má prístup niekde, kde sa nachádzajú cenné informácie.

[2] Jedná sa o proces prerušenia inštrukcie volania funkcie a vykonanie vlastných činností či už pred alebo po vykonaní pôvodnej funkcie. Pôvodná funkcia však taktiež nemusí byť vôbec spustená. V prípade *sandboxov* sa najčastejšie modifikuje cieľ skoku za účelom zaznamenania systémového volania.

Prístup k súborom je tiež cenná informácia v prípade, že sa autor kódu rozhodne importovať symboly počas behu programu. Táto analýza môže odhaliť dodatočne importované symboly, mimo tých, ktoré boli zistené pri statickej analýze.

3.2.2 Sieťová komunikácia

V prípade sieťovej komunikácie je ideálne zachytávať kompletne všetku komunikáciu, ale je potrebné vedieť na čo sa sústrediť pri jej následnej analýze. Dôraz kladieme na určité protokoly, ktoré sú najvýznamnejšie z pohľadu toho, čo by škodlivý kód mohol vykonávať.

Protokol DNS — Domain Name System — je sieťový protokol, ktorý dokáže na vyžiadanie zistiť IP adresu cieľovej stanice, pokiaľ poznáme len jej doménové meno. Klient tohto protokolu je implementovaný priamo v operačnom systéme a využíva ho napríklad funkcia zo socket API, `gethostbyname`, ktorá sa nachádza v kóde takmer každého programu využívajúceho socket API. Tento protokol je pre nás preto zaujímavý, lebo v prípade, že sa auto *malwaru* nespolieha, že jeho server bude stále na jednej IP, tak použije doménové meno.

Ďalším zaujímavým protokolom je HTTP, ktorým je možné napríklad preniesť obsah súboru, čo sa často v *malwari* robí a taktiež existujú funkcie vo Windows API[3], ktoré dokážu odosielať a prijímať HTTP požiadavky resp. odpovede s minimálnymi technickými znalosťami. Na obrázku 3.2 zobrazené využitie HTTP protokolu *malwarom* Adylkuzz na prijímanie príkazov od kontrolného C&C (angl. *Command and Control*) serveru.

IRC — Internet Relay Chat — je protokol určený na textovú komunikáciu medzi niekoľko učastníkmi naraz. Jeho použitie v *malware* je známe napríklad pri C&C *botnetoch*[4], kedy je možné zadávať príkazy všetkým cez zaslané správy.

```
GET /mine.txt HTTP/1.1
Connection: close, TE
TE: trailers
User-Agent: LuaSocket 3.0-rc1
Host: 08.super5566.com

HTTP/1.1 200 OK
Server: nginx
Date: Wed, 17 May 2017 15:47:41 GMT
Content-Type: text/plain
Content-Length: 158
Last-Modified: Tue, 16 May 2017 08:38:16 GMT
Connection: close
ETag: "591aba78-9e"
Accept-Ranges: bytes

-a cryptonight -o stratum+tcp://xmr.crypto-pool.fr:443 -u 48np7fEXZBwPVzhDk5MeZoai4iLAAharXK62ziZe8SFpdmGW87n8GHoTxC5RftYLqwQNaSUjj5bHvXUTVBWgsm7PTBW7xM3 -p x
```

Obr. 3.2: Ukážka použitia protokolu HTTP v *malwari* Adylkuzz.

3.2.3 Registre

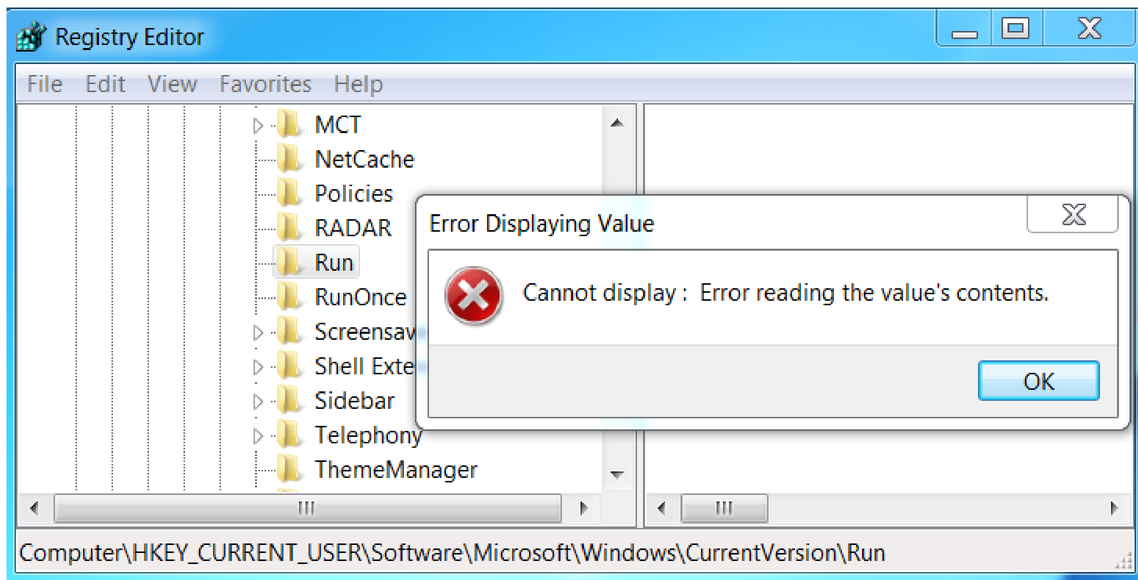
Na platforme Windows slúžia registre ako databáza pre nastavenie operačného systému. Fungujú na princípe stromovej hierarchie, ktorá má jeden spoločný koreň. Tento model je veľmi podobný klasickému súborovému systému. Namiesto súborov sa však na listovej úrovni tohto stromu nachádzajú páry kľúč–hodnota. Pomocou registrov je možné vykonávať mnoho činností potenciálne zneužiteľných *malwarom*, ako napríklad zoznam programov, ktoré sa spustia pri spustení systému. Pre analyzátora je preto podstatné, ku ktorým registrom daný program pristupoval, menil a ktoré mazal.

Autori *malware* stále hľadajú spôsoby ako ich *malware* môže získať perzistenciu a pritom zostať v utajení. Na obrázku 3.3 je zobrazený *malware* Kovter, ktorý celé svoj telo ukladá

[3]Funkcie ako `HttpOpenRequest`, `HttpSendRequest`, `HttpQueryInfo` a mnoho iných z `wininet.dll`.

[4]Sieť prepojených zariadení, riadených treťou stranou, najčastejšie bez vedomia vlastníka zariadenia.

do registrového kľúča. Pomocou špeciálneho názvu následne spôsobí, že obsah registrov je potom nezobraziteľný bežným nástrojom `regedit` v prostredí Windows.



Obr. 3.3: Ukážka toho, čo spôsobí prítomnosť *ransomwaru* Kovter.

3.2.4 Pomenované objekty

Pomenovanými objektami na platforme Windows sa najčastejšie myslia synchronizačné resp. IPC (angl. *inter-process communication* – medziprocesová komunikácia) prostriedky ako sú mutexy, semafóry, eventy, atómy a mnoho iných. Pri vytváraní takéhoto objektu mu dávame unikátne meno. Pokiaľ však objekt s daným menom už existuje, tak ho zdieľame s iným procesom. Tieto unikátne mená tým pádom môžu prezradiť o aký software sa jedná, pretože tieto mená musí vyplniť sám programátor a nemajú preto tendenciu sa často meniť.

Hlavnou úlohou pomenovaných objektov v rámci *malware* je zabrániť opätovnému spusteniu programu. To je tiež ďalším dôvodom prečo ich skúmame. Unikátna inštancia procesu je pomerne bežná vlastnosť *malware* a pomenované objekty sú bežný spôsob, ako to dosiahnuť. Na obrázku 3.4 je *ransomware* Crypt0L0cker, ktorý túto techniku využíva.

Detekcia pomocou názvov mutexov je pomerne nová metóda, ktorá je praxi aspoň zatiaľ spoľahlivá. Je preto vhodné sa pri dynamickej analýze zamerať na mutexy.

```
// WinMain
// ...
// CreateMutexW(0, 0, L"Global\otozidysotohesaqojegihi")
if (callFunctionByHash(HASH_CREATE_MUTEX_W, wString))
    exit(1);
```

Obr. 3.4: *Ransomware* Cryp0L0cker, ktorý sa ukončí ak už existuje rovnaký mutex.

Kapitola 4

Nástroje na analýzu v spoločnosti AVG Technologies

Táto kapitola popisuje niektoré nástroje, ktoré sa používajú v spoločnosti AVG Technologies na analýzu *malwaru*. Popísané sú iba tie nástroje, ktoré sú kľúčové pre túto prácu. Na ďalej spomenutých nástrojoch, či systémoch, táto práca zakladá a prípadne ich aj vylepšuje.

4.1 Clusty

Clusty je novovznikajúci systém na zhlukovú analýzu (angl. *clustering*). Jeho vstupom sú vzorky z AVG databáze, o ktorých zatiaľ nie je jasné, či sa jedná o *malware*, čistý program alebo o tzv. PUP (angl. *Potentionally Unwanted Program* – potenciálne nechcený program). Motiváciou, prečo takýto systém vyvíjať, je zefektívnenie práce analytika *malwaru*, ktorý namiesto analyzovania jednotlivých vzoriek môže analyzovať kompletne celé zhluky, čo povedie k urýchleniu celého procesu analýzy. Cieľom je automatizovane hľadať podobnosti medzi jednotlivými súbormi a na základe toho vytvárať zhluky súborov. Fungovanie môžeme rozdeliť na tri fázy — analýza, *clustering* a extrakcia spoločných vlastností. Jednotlivé fázy si popíšeme podrobnejšie na nasledujúcich stranách.

Analýza

Pri analýze dochádza k predspracovaniu analyzovaného súboru a k zberu informácií o ňom. Do predspracovania rátame spúšťanie série *unpackerov*, ktoré sa snažia odstrániť *packery* a umožniť tak statickú analýzu. K zberu informácií sa používa niekoľko zdrojov.

- Nástroj *fileinfo* popísaný v podkapitole 4.2.
- Detekcia ostatných antivírov pomocou služby VirusTotal [16].
- Fuzzy hash – *ssdeep* [7].
- V prípade spustiteľného súboru pre *.NET framework* sa použije *.NET disassembler* na extrakciu informácií o *CLR* objektoch.
- V prípade spustiteľného súboru pre Android sa použije APK analyzátor [35].
- Pokiaľ je na službe VirusTotal dostupný *Cuckoo JSON report*, tak sa stiahne. Viac o Cuckoo je popísané v podkapitole 4.3.

Nakoniec dôjde k rozdeleniu súborov do kategórií ako sú PE, ELF, .NET a iné.

Clustering

Clustering sa vykonáva nad každou kategóriou zvlášť. Každá kategória má iné rysy na základe ktorých sa *clusteruje*[1]. Rysy sú pritom zoradené od tých najviac prioritných po tie najmenej prioritné a každý súbor je priradený výhradne do jedného zhluku. Na to aby vznikol niektorý z *clusterov* je potrebné aby mal istú minimálnu veľkosť, ktorá je pre každý rys iná. U niektorých rysov sa pritom hladí na exaktnú zhodu, u iných sa hľadí na podobnosť. Niektoré z rysov pre kategóriu PE, zoradené podľa priority, sú nasledovné.

- Exaktná zhoda tabuľky symbolov
- Exaktná zhoda cesty k PDB súboru
- Exaktná zhoda mien mutexov
- Exaktná zhoda stromu *resourcov*
- Podobnosť ($< 100\%$) stromu *resourcov*
- Exaktná zhoda tabuľky importovaných symbolov

Extrakcia spoločných vlastností

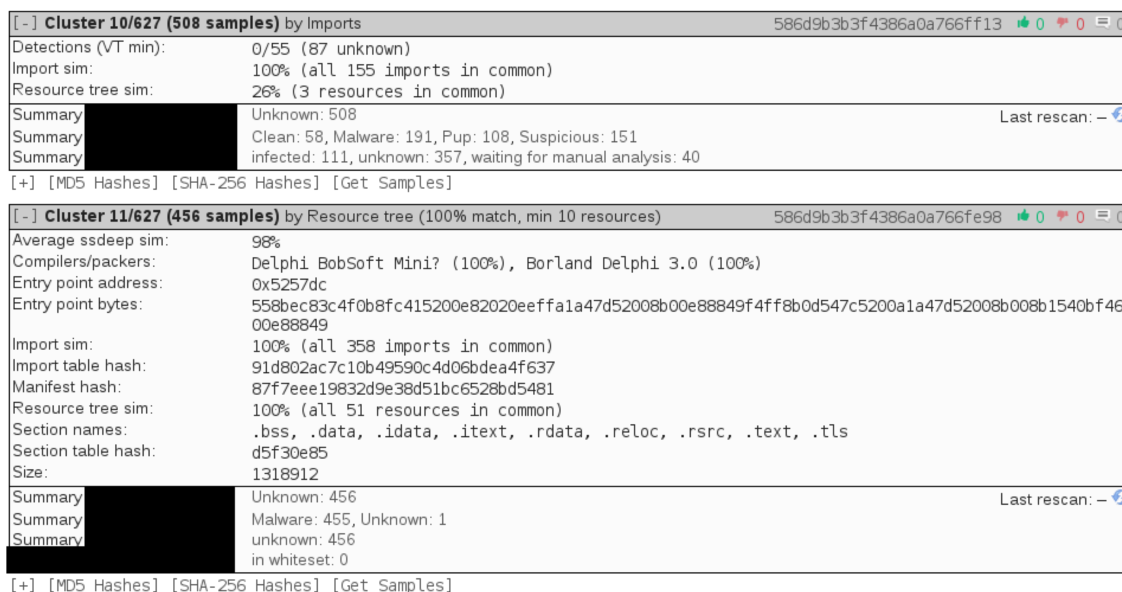
Zo vzoriek zo vzniknutých *clusterov* sa ako posledný krok vyextrahujú niektoré vybrané vlastností, ktoré sa priradia samotnému *clusteru*. Tie sú rozdielne pre každú kategóriu. U každej vlastnosti je navyše určené, či sa k danému *clusteru* priradí pokiaľ je u všetkých vzoriek rovnaká, alebo sa len vypočíta podobnosť tejto vlastnosti. Niektoré z vlastností u formátu PE sú nasledovné.

- Zoznam detekovaných prekladačov
- Adresa vstupného bodu
- Sekvencia bajtov na vstupnom bode
- Podobnosť importovaných symbolov
- Názvy sekcií

Výsledok sa nakoniec uloží do databázy, odkiaľ je možné zobraziť všetko o *clusteroch* pomocou webového rozhrania, ktoré je možné vidieť na obrázku 4.1, kde sú zobrazené dva *clustery*. Prvý vznikol na základe exaktnej zhody importovaných symbolov a jediná spoločná vlastnosť je 26% podobnosť *resourcov*. Druhý *cluster* vznikol na základne exaktnej zhody *resourcov* (ikony, obrázky atď.). Spoločných vlastností je v tomto prípade oveľa viac a to napríklad zhoda detekovaného použitého prekladača, zhodnosť adresy vstupného bodu a bajtov na vstupnom bode, absolútna zhoda importovaných symbolov a mnoho iných.

Clusty taktiež poskytuje RESTful API cez HTTP protokol, pomocou ktorého je možné získať informácie o jednotlivých *clusteroch* vo formáte JSON a ďalej spracovávať pre vlastné účely.

[1] Slovenský preklad je zhlukuje, avšak pre odlíšenie technického významu slova v tomto kontexte sa uvádza anglická varianta



Obr. 4.1: Webové rozhranie nástroja Clusty.

4.2 Fileinfo

Fileinfo je nástroj, ktorý vznikol v rámci projektu rekonfigurovateľného spätného prekladača RetDec [14]. Jedná sa o nástroj na statickú analýzu spustiteľných súborov. Extrahuje statické informácie zo spustiteľného súboru do unifikovanej reprezentácie a tie prezentuje na výstup. Tento nástroj využíva práve Clusty na extrakciu rysov na základe ktorých robí zhlukovú analýzu. *Fileinfo* aktuálne podporuje analýzu formátov PE, ELF, COFF, Mach-O a Intel HEX. Niektoré zo statických informácií, ktoré *fileinfo* dokáže extrahovať sú:

- Základné informácie z hlavičky formátu spustiteľného súboru
  - Architektúra
  - Endianita architektúry
  - Trieda súboru – 32-bit/64-bit
  - Typ súboru – spustiteľný súbor, statická/dynamická knižnica, objektový súbor
  - Adresa vstupného bodu
  - Sekvencia bajtov na vstupnom bode
- Tabuľka sekcií
  - Názov
  - Fyzická adresa a veľkosť
  - Logická adresa a veľkosť
  - Charakteristika
- Tabuľka importovaných symbolov
- Tabuľka exportovaných symbolov

- Tabuľka relokácií
- Tabuľka symbolov
- Informácie z *loaderu* spustiteľných súborov

Aj napriek tomu, že je výstup unifikovaný pre všetky podporované formáty spustiteľných súborov, tak každý formát obsahuje nejakú špecifickú vlastnosť, ktorú unifikovať nie je možné. Preto sa pri rôznych formátoch môžeme stretnúť s dodatočnými informáciami, ktoré sú unikátne pre daný formát. U formátu PE sú to nasledovné informácie.

- *Rich header* – špeciálna nedokumentovaná hlavička obsahujúca informácie o použitom prekladači, pokiaľ bol použitý prekladača rodiny Microsoft Visual C++
- Tabuľka tzv. *resourcov* (ikony, menu, obrázky, informácie o verzií aplikácie atď.)
- Manifest vo formáte XML

Hlavnou a dôležitou časťou tohto nástroja je knižnica *fileformatl*, vyvinutá taktiež pre rekonfigurovateľný spätný prekladač RetDec [14] v rámci bakalárskej práce M. Zavorala [41]. Slúži na parsovanie obsahu spustiteľných súborov a ich transformáciu do unifikovanej internej reprezentácie. Pre formát PE používa knižnicu tretej strany, nazývanú *pelib* [13]. Nástroj *fileinfo* potom len použije internú reprezentáciu knižnice *fileformatl* a prezentuje ju v rôznych podobách formou tzv. prezentérov. Aktuálne existujú dva prezentéry a to *plain* prezentér vypisujúci informácie v čitateľnej podobe a JSON prezentér určený pre strojové spracovanie výstupu.

Výstup nástroja *fileinfo* s použitím *plain* prezentéru vyzerá tak, ako je to znázornené na ukážke 4.1. Pokiaľ je však *fileinfo* spustené v tzv. *verbose* móde, tak vypisuje podstatne viac informácií o danom súbore.

Kód 4.1: Výstup nástroja *fileinfo*

```
Input file                  : helloworld.exe
CRC32                       : 3b4de344
MD5                         : bbc5008952666b91ee0b7f6ed9b46475
SHA256                      :
    ↪ 07BC6FDA0AD764970AF72F4759DD1517F2F28D60BC7049B88E9127C7A4DE89CE
File format                 : PE
File class                  : 32-bit
File type                   : Executable file
Architecture                : x86
Endianness                  : Little endian
Image base address          : 0x400000
Entry point address         : 0x41104b
Entry point offset          : 0x44b
Entry point section name    : .text
Entry point section index: 1
Bytes on entry point        :
    ↪ e9900f0000e99f3c0000e9363d0000e9d10a0000e93c250000e907370000e9a2120000e9cd1b
    ↪ 0000e998230000e98d3b0000
Detected compiler/packer : Visual Studio 2005/2008/2010 Visual C++ (14/15/16)
    ↪ (Win32 Console Application with Precompiled Headers in Debug Mode. Debug
    ↪ Information Format: Program Database for Edit & Continue (/ZI).) (100%) (120
    ↪ from 120 significant nibbles)
Detected compiler/packer : Visual C++ (8.0 (Debug)) Microsoft (100\%) (42 from 42
    ↪ significant nibbles)
Rich header offset          : 0x80
Rich header key             : 0x469eb28c
Rich header signature       :
    ↪ 00ef9cb40000000201015e3b0000000301055e3b0000001701045e3b0000000d00cbffdd
    ↪ 000000200010000
                              0000004101045e970000000100ff5e920000000101025e9700000001
```

4.3 Cuckoo

Cuckoo je *open-source* nástroj na dynamickú analýzu spustiteľných súborov [2]. V rámci AVG Technologies je to jeden z nástrojov na zisťovanie dynamických vlastností analyzovaných programov. Infraštruktúra *sandboxu* sa skladá z riadiaceho stroja nazývaného *host* (hostiteľ) a z množiny virtuálnych strojov, ktoré sa nazývaju *guesti* (hostia). *Host* prijíma požiadavky na spustenie programov a tie smeruje na *guestov* na ktorých sa v kontrolovanom prostredí spúšťajú dané programy. Po tom čo program skončí svoju činnosť sa z *guesta* pošle tzv. *raw report* (surová správa) na *hosta*. Ten následne vyhodnotí daný *raw report* a vygeneruje *report* vo formáte JSON.

Na *guestoch* bežia analyzéry. Tie definujú ako spúšťať jednotlivé druhy analyzovaných súborov[2] a taktiež definujú dodatočné operácie, ktorá sa vykonávajú paralelne spolu s vykonávaním analyzovaného súboru. Medzi dodatočné operácie patrí napríklad vyhotovovanie *screenshotov*, simulácia pohybu myšy a stlačení kláves, produkovanie falošných informácií do *sandboxu* aby sa zamaskovalo čisté prostredie a iné.

Ďalšou dôležitou časťou je *monitor*, ktorý sa taktiež nachádza na *guestoch*. Jeho úlohou je pomocou techniky nazývanej *DLL injekcia* [36] (angl. *DLL injection*) presmerovať volania Windows API cez kód, ktorý zaznamená hodnoty parameterov týchto volaní a až následne spustí odpovedajúce volanie vo Windows API. *Monitor* taktiež implementuje mnoho dodatočných heuristík, ktoré simulujú čakanie pomocou funkcií na uspanie procesu a iné.

Príklad jednoduchého *JSON reportu* je možné vidieť na ukážke 4.2. V skutočnosti toho obsahuje takýto *report* omnoho viac, avšak uvedená je len časť toho najpodstatnejšieho z pohľadu tejto práce.

Kód 4.2: Ukážka Cuckoo *JSON reportu*

```
{
  "network": {
    "http": [
      {
        "uri": "http://go.microsoft.com/fwlink/?LinkId=121315",
        "method": "GET"
      }
    ],
    "domains": [
      {
        "ip": "104.81.213.49",
        "domain": "go.microsoft.com"
      }
    ]
  },
  "behavior": {
    "summary": {
      "files": [
        "C:\\Windows\\System32\\MSCOREE.DLL.local"
      ],
      "keys": [
        "HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\.NETFramework\\Policy\\v4.0"
      ],
      "events": [
        "Global\\CLR_PerfMon_StartEnumEvent"
      ]
    }
  }
}
```

[2] Cuckoo podporuje rôzne druhy súborov, nie len tie spustiteľné. Je možné si vyžiadať analýzu súborov vo formáte `doc`, `xls`, `js`, `pdf` a mnoho iných.

4.4 YARA

YARA [18] označuje deklaratívny jazyk na popis vzorov v súboroch, ale taktiež označuje nástroj, ktorý je schopný tieto vzory v súboroch vyhľadávať. Projekt YARA je spravovaný tvorcami projektu VirusTotal. V rámci AVG Technologies sa YARA používa na detekciu vzorov, ktoré klasifikujú škodlivé súbory.

Jazyk YARA

YARA ako jazyk pozostáva z pravidiel, pričom každé pravidlo popisuje nejaký vzor. Každé pravidlo môže byť označené tzv. *tagom*. Tento *tag* je možné použiť pre označenie daného pravidla do nejakej rodiny, ktorú je možné pri vyhľadávaní daného vzoru vyfiltrovať. Základná štruktúra každého pravidla je nasledovná:

- Metadáta
- Reťazce
- Podmienka

Metadáta a reťazce nie sú povinné, avšak každé pravidlo musí mať definovanú podmienku, ktorá pokiaľ je vyhodnotená ako pravdivá, tak sa pravidlo považuje za nájdené vo vstupnom súbore.

Metadáta nemajú žiaden špeciálny účel a pri vyhľadávaní vzoru nie sú nijak použité. Slúžia len na uloženie užívateľských dát k danému pravidlu. Metadáta sú definované ako kľúč–hodnota, kde hodnota môže byť celočíselného typu, reťazcový literál alebo booleovská hodnota.

Reťazce sú definované pomocou unikátneho identifikátoru a samotného reťazca. Identifikátor musí začínať znakom `$` za ktorým nasleduje postupnosť alfanumerických znakov. Reťazce môžu byť troch rôznych typov.

- Textový reťazec
- Hexadecimálny reťazec
- Regulárny výraz

Textové reťazce sa skladajú výhradne z tlačiteľných znakov. Jedná sa len o ASCII alebo Unicode reťazce u ktorých je maximálne možné špecifikovať, či sa pri vyhľadávaní majú ignorovať malé/veľké písmená, alebo či sa jedná o exaktnú zhodu reťazca.

Hexadecimálne reťazce pozostávajú priamo z oktetov. Jedná sa o postupnosť surových bajtov. Túto postupnosť je možné definovať aj pomocou niekoľko pomocných operátorov rozširujúcich vyjadrovaciu schopnosť. Tieto operátory sú nasledovné.

- `?` – *wildcard*, značí ľubovolný polbajt `0-F`
- `X|Y` – *alternatíva*, značí alternatívu medzi postupnosťou bajtov `A` alebo `B`
- `[N-M]` – *skok*, značí ľubovolných `N` až `M` bajtov cez ktoré je možné preskočiť pričom `N` aj `M` môžu byť vynechané pre neobmedzený počet

Regulárne výrazy v jazyku YARA odpovedajú regulárnym výrazom v programovacom jazyku Perl. Nepodporujú však niektoré vymoženosti ako sú napríklad spätné odkazy (angl. *back references*), zachytávajúce skupiny (angl. *capturing groups*), či POSIXové skupiny znakov.

Názornú ukážku pravidla zapísaného v jazyku YARA je možné vidieť na ukážke kódu 4.3. Podmienka tohto pravidla vraví, že každý z definovaných reťazcov sa musí v súbore nachádzať minimálne 2 krát a jeho pozícia v súbore musí byť minimálne 100.

Kód 4.3: Ukážkové YARA pravidlo.

```
rule example_rule : first_tag second_tag
{
  meta:
    int_meta = 42
    str_meta = "Value"
    bool_meta = true
  strings:
    // Text string
    $text = "Text␣string" ascii wide
    // Hexadecimal string
    $hex = { AB C? ?? [4-6] ( FF | EE ) }
    // Regular expression
    $regex = /md5: [0-9a-zA-Z]{32}/
  condition:
    for all of ($*) : ( # >= 2 and @ >= 100 )
}
```

Nástroj YARA

Nástroj YARA slúži na výhľadávanie vzoru zapísaného vo forme YARA pravidiel v súboroch. Štandardne sa jedná o ľubovolné súbory, či už binárne alebo textové. Výstupom takéhoto vyhľadávania je zoznam pravidiel, u ktorých bola podmienka splnená.

Obsah spustiteľných súborov však pri takomto vyhľadávaní stráca sémantiku. Všetko je chápané len ako postupnosť bajtov. Z tohoto dôvodu vznikli tzv. YARA *moduly*. Modul dokáže rozparsovať vstupný súbor a pridať dodatočnú sémantiku dátam. Každý modul zároveň sprístupní do YARA jazyka túto sémantiku vo forme štruktúry s unikátnym menom modulu. Pridávanie sémantiky vstupnému súboru však nie je jediné užitie modulov. Moduly môžu pridávať ľubovolné funkcie, či konštanty, ktoré je možné využiť pri písaní pravidiel. Nástroj YARA v základe obsahuje niekoľko modulov.

- Modul `pe` na parsovanie spustiteľných súborov vo formáte PE.
- Modul `elf` na parsovanie spustiteľných súborov vo formáte ELF.
- Modul `math` obsahujúci implementáciu funkcií ako počítanie entropie, priemernej hodnoty, odchýlky, či autokorelácie.
- Modul `hash` pridávajúci funkcie na počítanie MD5, SHA1, SHA256 a CRC32.
- Modul `magic`, ktorého pridáva jedine funkciu `type()`. Táto funkcia dokáže na základe tzv. magickej hodnoty na začiatku súboru určiť, o aký druh súboru sa jedná. Napríklad pre PDF dokument by funkcia vrátila `"PDF"`.

- Modul `cuckoo`, ktorý nepatrí medzi klasické moduly. Tento modul totižto nijak nepracuje so vstupným súborom, v ktorom vyhľadávame, ale pracuje s *reportom* sandboxu Cuckoo, ktorý bol popísaný v podkapitole 4.3. Parsuje sa výstup sandboxu a sprístupňuje sa vo forme YARA funkcií použiteľných v YARA pravidlách. Použitím tohto modulu pridávame k statickej analýze, ktorú YARA predstavuje aj dynamickú analýzu.

Ukážka pravidla využívajúceho moduly je možné vidieť na ukážke 4.4. Dané pravidlo bude vyhodnotené ako pravdivé, pokiaľ bude počet sekcií v PE súbore minimálne 3, MD5 hash tretej sekcie bude mať určitú hodnotu a počas svojho behu tento program pristúpil k súboru na ceste `c:\file.txt`.

Kód 4.4: YARA pravidlo využívajúce moduly.

```
import "cuckoo"
import "hash"
import "pe"

rule example_rule_with_module
{
  condition:
    (pe.number_of_sections >= 3) and
    (hash.md5(pe.sections[2].raw_data_offset, pe.sections[2].raw_data_size)
        ↪ == "900150983cd24fb0d6963f7d28e17f72") and
    cuckoo.filesystem.file_access(/c:\\file\.txt$/)
}
```

Projekt YARA taktiež umožňuje jednoduchú integráciu do vlastného kódu. Celkovo je totiž hlavná funkcionalita nástroja YARA implementovaná vo forme knižnice `libyara`. Samotný nástroj `yara` je len konzolové rozhranie nad touto knižnicou. To nám umožňuje vyhľadávanie YARA pravidlá vo vlastnej réžií.

Kapitola 5

Návrh nových analýz a generovania detekčného vzoru

Táto kapitola popisuje návrh nových analýz, ktoré použije ako systém Clusty, popísaný v podkapitole 4.1, tak aj nástroj na generovanie detekčného vzoru vo forme YARA pravidla. V druhej časti tejto kapitoly je popísaný návrh samotného nástroja.

5.1 Nové analýzy

Motiváciou návrhu nových analýz je poskytnúť lepšie základy pre tvorbu detekčného vzoru. Je dôležité sa sústrediť na analýzy, ktoré dokážu poskytnúť čo najunikátnejší popis množiny súborov. Na tie bol kladený dôraz v kapitole 2.

5.1.1 Certifikáty a podpis

Pred započatím tejto práce knižnica *fileformatl* nepodporovala spracovanie certifikátov, takže táto informácia nebola nijak dostupná cez jej rozhranie. Jedná sa však o informáciu, ktorá ponúka dodatočné informácie o danom spustiteľnom súbore. Zároveň nám overením podpisu umožňuje overiť to či bol daný súbor nejak modifikovaný a to je pri analýze *malwaru* ďalšia cenná informácia. Táto nová analýza spočíva v prezentácií informácií o jednotlivých certifikátoch v spustiteľnom súbore a následnom overení podpisu obsahu PE súboru.

Ako bolo spomínané v kapitole 2, tak PE súbor, ktorý obsahuje *security directory* má v sebe uloženú PKCS7 štruktúru s podpísanými dátami a s X.509 certifikátmi zakódovanú vo formáte DER. Z jednotlivých certifikátov je potom nutné vybrať podstatné informácie, ktoré odprezentovať na výstupe. Medzi tieto informácie zaradíme:

- Meno subjektu
- Organizácia subjektu
- Celková identifikácia subjektu
- Meno vystavovateľa
- Organizácia vystavovateľa
- Celková identifikácia vystavovateľa

- Algoritmus pre verejný kľúč
- Algoritmus pre podpis (kombinácia algoritmu, ktorým sa vypočíta hash podpisovaného obsahu a ktorým sa hashovaný obsah podpíše)
- Sériové číslo
- Dátum platnosti (od-do)

Ďalej je vhodné vypočítať hash samotného certifikátu v DER kódovaní. Tento hash je možné použiť v databáze AVG Technologies, pre vyhľadávanie známych certifikátov. Vypočítame ako SHA1, tak aj SHA256.

Nakoľko sa v PKCS7 štruktúre nachádza celý reťazec certifikátov, tak by bolo vhodné ich odprezentovať všetky a v rozumnom poradí. Pokiaľ sa v PKCS7 štruktúre nachádza protipodpis, tak sa medzi certifikátmi nachádzajú dva reťazce. Prvý z nich začína certifikátom podpisovateľa a ten druhý certifikátom protipodpisovateľa. Rekonštrukciu reťazcov je možné robiť nasledovne:

1. Zvolíme si certifikát podpisovateľa.
2. Zoberieme meno vystavovateľa zvoleného certifikátu a hľadáme certifikát s daným subjektom.
   (a) Ak ho nájdeme, tak ho zvolíme.
   (b) Ak ho nenájdeme, tak buďto:
      i. Rekonštruujeme prvý reťazec a existuje protipodpis. Potom zakončíme prvý reťazec a zvolíme certifikát protipodpisovateľa.
      ii. Rekonštruujeme prvý reťazec a neexistuje protipodpis. Potom algoritmus končí.
      iii. Rekonštruujeme druhý reťazec. Potom algoritmus končí.
3. Opakujeme krok 2.

Pri overení podpisu sa taktiež pracuje so *security directory*, avšak už nepostačí len tieto dáta naparsovať a prezentovať užívateľovi, ale je nad nimi nutné robiť rôzne operácie. Overenie podpisu sa robí v niekoľko krokoch. Najskôr sa musí vypočítať hash obsahu PE súboru. Až získame tento hash, tak ho porovnáme s hashom uloženom v časti `contentInfo` PKCS7 štruktúry. Pokiaľ tieto hashe nesedia, tak nemusíme pokračovať ďalej, pretože je zjavné, že so súborom bolo manipulované a niekto jeho obsah modifikoval. Podpis tým pádom automaticky vyhodnotíme za neplatný. Pokiaľ ale tieto hashe budú rovnaké, tak ešte stále nemáme istotu, že so súborom sa nemanipulovalo. Autor *malwaru* mohol hash prepočítať sám a prepísať ho. Následne je preto nutné vypočítať hash z `contentInfo`. Pomocou verejného kľúča zo `signerInfo` časti PKCS7 štruktúry sa dešifruje podpísaný hash `contentInfo`, ktorý je tiež uložený v `signerInfo`. V prípade, že hash dešifrovaný a vypočítaný hash nie sú rovnaké, vyhodnotíme obsah ako neplatný, inak ako platný.

5.1.2 Rekonštrukcia .NET typov

Spôsob uloženia .NET informácií v súboroch formátu PE bol popísaný v podkapitole 2.5. Táto podkapitola bude rozčlenená na časti, ktoré popíšu spôsob rekonštrukcie všetkých potrebných typov.

Triedy

Triedy rozdeľujeme na tie, ktoré sú definované užívateľom a tie, ktoré boli importované z knižníc. Užívateľské triedy rekonštruujeme pomocou tabuľky `TypeDef`. V tejto tabuľke získame index do *string streamu*, kde sa nachádza názov triedy a menného priestoru. V prípade, že sa jedná o triedu s generickými parametrami, tak jej názov obsahuje špeciálnu časť, oddelenú znakom ‘ a nasledovanú počtom generických parametrov. Napríklad trieda `MyClass<T1,T2,T3>` má v *string streame* názov `MyClass‘3`. Táto špeciálna časť sa oddelí a počet generických parametrov sa poznamená. Trieda, ktorá ma názov `<Module>` sa ignoruje, pretože sa v skutočnosti nejedná o triedu.

Ďalšie vlastnosti definovanej triedy rozlíšime na základe bitovej masky `Flags` v zázname tabuľky `TypeDef`. Podstatné bitové masky sú znázornené v tabuľke 5.1.

Názov	**Bitová maska**	**Popis**
`TypeNotPublic`	0x00000000	Trieda je označená ako `private`.
`TypePublic`	0x00000001	Trieda je označená ako `public`.
`TypeNestedPublic`	0x00000002	Trieda je označená ako `public`.
`TypeNestedPrivate`	0x00000003	Trieda je označená ako `private`.
`TypeNestedFamily`	0x00000004	Trieda je označená ako `protected`.
`TypeClass`	0x00000000	Jedná sa o `class`.
`TypeInterface`	0x00000020	Jedná sa o `interface`.
`TypeClassAbstract`	0x00000080	Trieda je označená ako `abstract`.
`TypeClassSealed`	0x00000100	Trieda je označená ako `sealed`.

Tabuľka 5.1: Tabuľka s bitovými maskami pre `Flags` u `TypeDef` tabuľky.

Importované triedy síce nechceme generovať do výstupu, ale sú nutné pre rekonštrukciu dedičnosti. Tie získame rovnako ako užívateľské triedy avšak z tabuľky `TypeRef`. U importovaných tried neexistujú žiadne príznaky.

Metódy

K rekonštrukcií metód jednotlivých tried nám pomôže tabuľka `MethodDef`. Každý záznam v `TypeDef` tabuľke obsahuje v atribúte `MethodList` odkaz do `MethodDef` tabuľky. Počet metód je nutné dopočítať z rozdielu dvoch po sebe idúcich záznamov `TypeDef` tabuľky. V prípade posledného záznamu sa robí rozdiel veľkosti `MethodDef` tabuľky a `MethodList` atribútu tohto záznamu. Metódy s názvami `.ctor` a `.cctor` sú označené ako konštruktory.

Kvalifikátory použité pri definícií metódy sú rekonštruované z `Flags` atribútu. Možné hodnoty bitovej masky sú zobrazené v tabuľke 5.2.

Počet a typy parametrov metódy je nutné zistiť zo signatúry zapísanej v *blob streame* indexovanej atribútom `Signature`. Tá obsahuje v prvom bajte nastavenú bitovú masku `0x20` pokiaľ sa jedná o generickú metódu. Nasleduje zapísaný počet parametrov a sekvencia identifikátorov typov jednotlivých parametrov. Mená parametrov sa rekonštruujú z tabuľky

Názov	Bitová maska	Popis
`MethodPrivate`	`0x0001`	Metóda je označená ako `private`.
`MethodFamily`	`0x0004`	Metóda je označená ako `protected`.
`MethodPublic`	`0x0006`	Metóda je označená ako `public`.
`MethodStatic`	`0x0010`	Metóda je označená ako `static`.
`MethodFinal`	`0x0020`	Metóda je označená ako `final`.
`MethodVirtual`	`0x0040`	Metóda je označená ako `virtual`.
`MethodAbstract`	`0x0400`	Metóda je označená ako `abstract`.

Tabuľka 5.2: Tabuľka s bitovými maskami pre `Flags` u `MethodDef` tabuľky.

`Param` do ktorej odkazuje atribút `ParamList`. Záznamy tejto tabuľky taktiež obsahuje atribúty `Flags`, ktorých možné hodnoty sú naznačené v tabuľke 5.3.

Názov	Bitová maska	Popis
`ParamOut`	`0x0002`	Parameter je označený ako `out`.

Tabuľka 5.3: Tabuľka s bitovými maskami pre `Flags` u `Param` tabuľky.

Atribúty

Tabuľka `Field` slúži na rekonštrukciu atribút jednotlivých tried. Záznam v tabuľke `TypeDef` sa odkazuje do tabuľky `Field` pomocou atribútu `FieldList`. Počet atribútov je určený na základe dvoch po sebe idúcich záznamov v tabuľke `TypeDef`. Pre posledný záznam tabuľky `TypeDef` sa robí rozdiel medzi veľkosťou tabuľky `Field` a atribútom `FieldList`.

U každého atribútu je možné zistiť jeho názov z atribútu `Name` odkazujúceho sa do *string streamu*. Ďalšie vlastnosti sú rekonštruované z atribútu `Flags`, ktorého možné hodnoty zobrazuje tabuľka 5.4.

Názov	Bitová maska	Popis
`FieldPrivate`	`0x0001`	Atribút je označený ako `private`.
`FieldFamily`	`0x0004`	Atribút je označený ako `protected`.
`FieldPublic`	`0x0006`	Atribút je označený ako `public`.
`FieldStatic`	`0x0010`	Atribút je označený ako `static`.

Tabuľka 5.4: Tabuľka s bitovými maskami pre `Flags` u `Field` tabuľky.

Vlastnosti

Na rekonštrukciu vlastností tried je použitá tabuľka `Property` obsahujúca samotné vlastnosti a tabuľka `PropertyMap`, ktorá asociuje záznam v tabuľke `TypeDef` so záznamami

v `Property`. Počet vlastností je nutné určiť z rozdielu atribútov `PropertyList` dvoch po sebe idúcich záznamov v tabuľke `PropertyMap`. Pre posledný záznam sa spraví rozdiel medzi posledným atribútom `PropertyList` a veľkosťou tabuľky `Property`.

Dedičnosť

Hierarchia tried sa rekonštruuje prechodom cez `TypeDef` tabuľku, kde atribút `Extends` sa odkazuje buďto do `TypeDef`, `TypeRef` alebo `TypeSpec` tabuľky. `TypeSpec` je použitá pokiaľ sa dedí z generickej triedy napríklad `Class<int>`. To ktorá sa použije sa rozozná podľa najnižších 2 bitov atribútu `Extends`. Skutočný index potom začína až od 2. bitu.

Prítomnosť len jediného `Extends` atribútu naznačuje, že nie je podporovaná viac násobná dedičnosť. Trieda však stále môže implementovať niekoľko rozhraní. Na to je ale potrebné použiť tabuľku `InterfaceImpl`. Atribút `Class`, odkazujúci sa na `TypeDef`, značí triedu implementujúcu rozhranie a atribút `Interface`, odkazujúci sa na `TypeDef`, `TypeRef` alebo `TypeSpec`, definuje ktoré rozhranie je implementované.

Generické parametre

U generických typov je vhodné rekonštruovať názvy jednotlivých generických typov, na čo sa použije tabuľka `GenericParam`. Atribút `Owner` sa odkazuje buďto na `TypeDef` alebo `MethodDef` podľa toho, či sa jedná o generický parameter triedy alebo metódy. Rozoznať je to možné najnižším bitom, pričom skutočný index potom začína od 1. bitu. `Name` sa odkazuje do *string streamu* a obsahuje samotné meno generického parametra.

Vnorené triedy

U vnorených tried je problém ten, že nemajú nastavený správny menný priestor v `TypeDef` zázname, resp. ich menný priestor je prázdny. Aby sme mohli správne určiť plne kvalifikovaný názov triedy, tak musíme poznať aj jej menný priestor. Tabuľka `NestedClass` obsahuje informácie o tom, ak sa jedna trieda nachádza v druhej. Atribút `NestedClass` je odkaz na vnorenú triedu a `EnclosingClass` na triedu, v ktorej sa nachádza.

TypeLib identifikátor

Lokalizovať *TypeLib* identifikátor v *string streame* nie je úplne triviálne. Najskôr je nutné v `TypeRef` tabuľke nájsť referenciu na triedu `GuidAttribute` importovanú z `mscorlib` modulu. Následne je nutné lokalizovať v tabuľke `MemberRef` záznam odkazujúci sa na index záznamu, ktorý sme našli v `TypeRef` tabuľke. Posledným krokom je nájdenie záznamu v tabuľke `CustomAttribute` odkaz na daný záznam v `MemberRef`. Tých môže byť niekoľko a preto je vždy nutné overiť, či sa daný záznam z `CustomAttribute` odkazuje na reťazec vyhovujúci regulárnemu výrazu spomenutom v kapitole 2.5.

5.1.3 Detekcia reťazcov

Detekcia reťazcov je ďalšia navrhnutá analýza do knižnice *fileformatl*. Reťazec chápeme ako postupnosť znakov, ktorá je zakončená netlačiteľným znakom[1]. Pri analyzovaní obsahu súboru je nutné si vymedziť, ktoré znaky chápeme ako súčasť reťazcov a ktoré ignorujeme. Ak totiž každú nenulovú sekvenciu chápeme ako reťazec, tak by sme detekovali mnoho

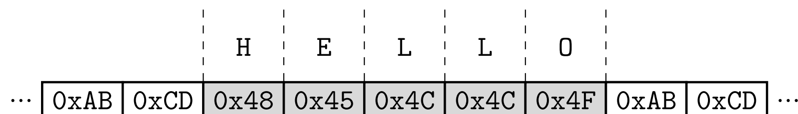
[1]Netlačiteľný znak je taký, ktorý vráti nulovú hodnotu pre funkciu jazyka C, `isprint`.

reťazcov, pričom väčšina z nich by žiaden reťazec v skutočnosti nebola. Našim cieľom je ideálne vyfiltrovať obsah tak, aby zostali len skutočné reťazce. Z toho dôvodu budeme za reťazec považovať len tie sekvencie, ktoré spĺňujú nasledovné podmienky.

- Každý bajt sekvencie je v rozmedzí ASCII hodnôt od `0x20` (medzera) po `0x7E` (`~`).
- Sekvencia má dĺžku aspoň 4 znaky.

Musíme si tiež určiť, kde tieto reťazce presne vyhľadávať. Automaticky môžeme vynechať hlavičky formátu, pretože reťazce v nich nie sú pre nás zaujímavé. Obmedzíme tým pádom vyhľadávanie len na sekcie. Isté sekcie však môžu obsahovať sekvencie, ktoré síce budú pripomínať reťazce, ale v skutočnosti to reťazce nebudú. Preto vynecháme sekcie, ktoré sú označené, že obsahujú kód.

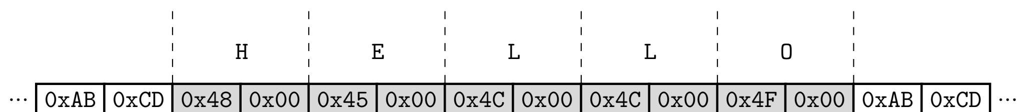
Ukážka možného reťazcu je naznačená na obrázku 5.1. Šedou farbou je znázornená časť, ktorá bola vyhodnotená ako obsah reťazca.



Obr. 5.1: ASCII reťazec `HELLO`.

Všetko, čo bolo spomenuté vyššie by však fungovalo len pre ASCII reťazce, ktoré používajú na jeden znak jeden bajt. V reálnom svete sa ale stretávame aj s tzv. *wide* (širokými) reťazcami, ktoré slúžia na uskladnenie Unicode reťazcov. U nich je na jeden znak potrebných niekoľko bajtov. Na platforme Windows sa stretávame s dvoma bajtami na znak, avšak všeobecne to môže byť aj štyri. Pri niekoľkých bajtoch sa musíme taktiež zaoberať endianitou, ktorú určíme z endianity formátu spustiteľného súboru, ktorý aktuálne spracovávame. Pre formát PE je to v drvivej väčšine prípadov *little-endian*, tj. najmenej významný bajt je uložený ako prvý.

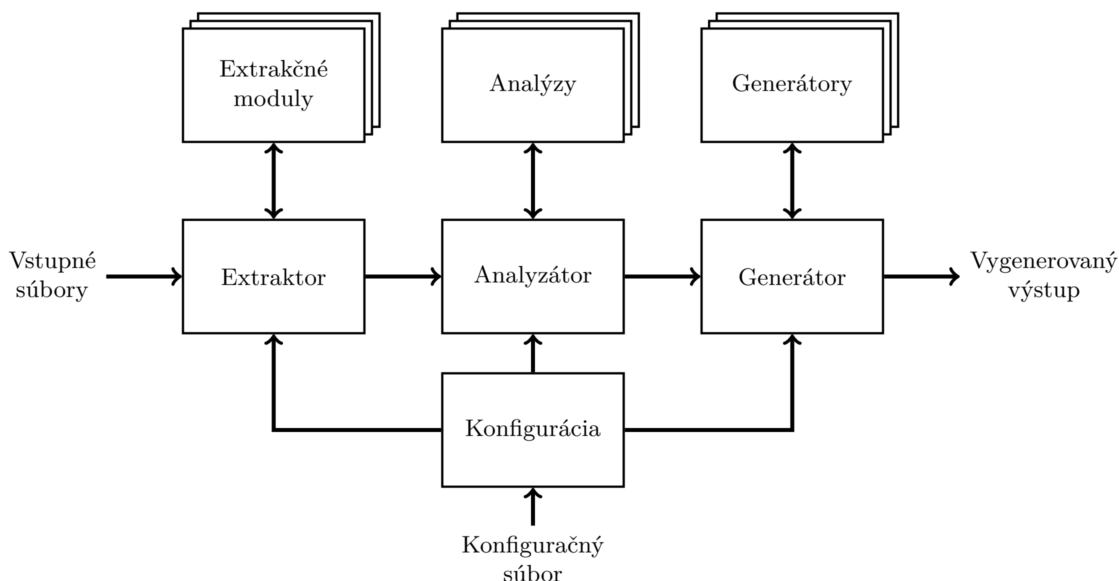
Pre jednoduchosť zatiaľ budeme uvažovať len o *wide* ASCII reťazcoch, ktoré síce používajú viac bajtov na jeden znak, ale uložený v nich je len znak v rozmedzí tabuľky ASCII. Rovnaký reťazec ako predtým, ale uložený ako *wide* reťazec je znázornený na obrázku 5.2. Pre tieto reťazce platia úplne rovnaké podmienky, avšak každý znak je rozšírený na viac bajtov.



Obr. 5.2: *Wide* reťazec `HELLO` s *little-endian* usporiadaním.

5.2 Generovanie detekčného vzoru

Pri návrhu nástroja na generovanie detekčného vzoru je braný ohľad hlavne na modulárnosť riešenia pre jednoduché spravovanie a budúce rozširovanie nástroja. Bloková schéma zobrazujúca princíp fungovania je načrtnutá na obrázku 5.3.



Obr. 5.3: Schéma nástroja na generovanie detekčného vzoru.

5.2.1 Extraktor

Extraktor zoberie množinu vstupných súborov a vykoná extrakciu informácií o každom súbore. Extraktor používa extrakčné moduly, ktoré poskytujú rozhranie na samotné získavanie informácií priamo zo súboru. Týmto spôsobom je možné vytvoriť niekoľko extrakčných modulov používajúcich rôzne metódy na extrakciu informácií, napríklad extrakcia priamo zo spustiteľného súboru, cez webové rozhranie, zo súboru obsahujúceho metadáta a rôzne iné. Aktuálne sa ráta len s jediným extrakčným modulom, ktorý používa knižnicu *fileformatl*.

5.2.2 Analyzátor

Analyzátor zoberie informácie, ktoré získal extraktor a použije ich v jednotlivých analýzach, ktoré spúšťa. Všetky analýzy fungujú na princípe frekvenčnej analýzy, tj. hľadajú, ktoré hodnoty pokrývajú ktoré súbory a vyhodnocujú histogram pokrytia jednotlivých súborov. Každá analýza má zoznam *pre* a *post* filtrov, ktoré určujú aké hodnoty môžu byť analyzované a ktoré hodnoty môžu byť oznámené na výstupe analýz. Výsledkom analyzátora je zoznam *skupín pokrytia*, tj. množina súborov, ktorá je pokrytá rovnakou množinou hodnôt. Všeobecný algoritmus je nasledovný:

1. Všetky vstupné súbory sú zaregistrované do analýzy.
2. Pre každú analyzovanú hodnotu každého súboru.
   (a) Analyzovaná hodnota je pridaná do konkrétnej analýzy spolu so súborom z ktorého pochádza.
   (b) Nad analyzovanou hodnotou je spustený zoznam *pre*-filtrov. Pokiaľ niektorý z filtrov zlyhá, teda vráti nepravdivú logickú hodnotu, tak je hodnota ignorovaná.
   (c) Pre danú hodnotu je poznamenané, že pokrýva daný súbor.
   (d) Ak neexistuje viac extrahovaných hodnôt pre aktuálny súbor, tak sa prechádza na ďalší.

3. Pre každú poznamenanú hodnotu.
   (a) Zoberie sa množina súborov, ktoré pokrýva a vytvorí sa *skupina pokrytia* z usporiadanej množiny pokrytých súborov.
   (b) Ak už existuje rovnaká *skupina pokrytia*, tak sa poznamenaná hodnota pridá do tejto skupiny.
   (c) Ak neexistuje, tak sa zaznamená táto *skupina pokrytia* ako nová skupina.
4. Nad *skupinami pokrytia* sú spustené *post*-filtre. Pokiaľ niektorý z filtrov zlyhá, tak je *skupina pokrytia* zmazaná.
5. Zvyšné *Skupiny pokrytia* sú usporiadané podľa mocnosti množiny pokrytých súborov.

Analýzy a ich konfigurácia

Všetky dostupné analýzy sú naznačené v tabuľke 5.5. Každá analýza navyše ponúka možnosti ako ju konfigurovať. Spoločné konfiguračné nastavenia sú minimálne percentuálne pokrytie a nutnosť pokrytia všetkých súborov. Pri minimálnom percentuálnom pokrytí je možné špecifikovať koľko percent súborov musí daná *skupina pokrytia* pokryť aby bola zarátaná do výsledku analýzy. Nutnosť pokrytia všetkých súborov zase špecifikuje, či prienik všetkých *skupín pokrytia* by zahrňoval všetky súbory registrované do analýzy. Vo vyššie spomenutej tabuľke vidíme aj štandardné nastavenia jednotlivých konfiguračných nastavení. Minimálne percentuálne pokrytia sú nastavené na 80% pre analýzy u ktorých nie je nutné pokrytie všetkých súborov, pretože je štandardne požadované aby analyzované hodnoty boli prítomné v drvivej väčšine analyzovaných súborov. U analýz, ktoré nemajú nutné pokrytie všetkých súborov je zase minimálne percentuálne pokrytie na 20% pretože tu nám postačuje, ak je to spoločná vlastnosť aspoň v malej podmnožine súborov. Hodnoty boli určené experimentálne.

Analyzovaná hodnota	**Pre formáty**	**Minimálne percentuálne pokrytie**	**Nutné pokrytie všetkých súborov**
Názov .NET triedy	.NET	80%	Nie
Názov .NET metód	.NET	80%	Nie
TypeLib identifikátora	.NET	0%	Áno
Detekované reťazce	PE	80%	Nie
Importované symboly (knižnica + symbol/ordinál)	PE	80%	Áno
Vstupný bod (adresa + bajty)	PE	0%	Áno
Sekcie (index + názov)	PE	80%	Nie
Cesta k PDB súboru	PE, .NET	0%	Áno
Rich hlavička	PE, .NET	0%	Áno
Podpisujúci	PE, .NET	0%	Áno

Tabuľka 5.5: Analýzy

Existuje ešte tretie spoločné nastavenie a to reštriktívnosť. Toto nastavenie je však štandardne vypnuté pre všetky analýzy a je ho možné zapnúť len na vyžiadanie. Restriktívnosť nemá žiaden vplyv na výsledok analýzy ale hrá veľkú rolu pri generovaní výsledkov, preto je toto konfiguračné nastavenie popísané neskôr.

Niektoré analýzy však majú aj špecifické konfiguračné nastavenia. Tie je možné vidieť spolu s ich štandardnými hodnotami v tabuľke 5.6.

Analyzovaná hodnota	**Dodatočné konfiguráčné nastavenia**
Názov .NET triedy	Min. dĺžka názvu: 4 Min. veľkosť *skupiny pokrytia*: 4
Názov .NET metód	Min. dĺžka názvu: 4 Min. veľkosť *skupiny pokrytia*: 4
Detekované reťazce	Min. dĺžka reťazca: 6 Min. veľkosť *skupiny pokrytia*: 4
Importované symboly	Min. veľkosť *skupiny pokrytia*: 4
Vstupný bod	Počet bajtov: 50
Sekcie	Min. veľkosť *skupiny pokrytia*: 4

Tabuľka 5.6: Špecifické nastavenia analýz

Biela listina

Primárne sa sústredíme v našom nástroji na *malware* a jeho detekciu. Mnoho krát sa však môže stať, že v spustiteľných súboroch budeme nachádzať hodnoty, ktoré sú napríklad bežne dopĺňané prekladačom a budú spoločné pre *malware*, ale aj pre čisté súbory. Tieto hodnoty by bolo vhodné eliminovať a vyhnúť sa tak zjavný chybám typu 1 (*false positive*). Z toho dôvodu do niektorých analýz zavedieme *whitelist* (biela listina), ktorá obsahuje zoznam hodnôt bežne sa vyskytujúcich v čistých súboroch a preto budú pri analýze ignorované. *Whitelist* je zavedený pre detekované reťazce, .NET metódy a sekcie. U .NET metód je *whitelist* vytvorený z najpočetnejších .NET metód z 15000 .NET súborov, u detekovaných reťazcov zo 100000 súborov z čistej sady databáze AVG Technologies a pre sekcie z vybraných najčastejších názvov sekcií [20].

5.2.3 Generátor

Posledná fáza tohto nástroja berie výstupy jednotlivých analýz a generuje z nich výstup. Zatiaľ sa počíta len s generovaním jedného YARA súboru obsahujúceho všetky pravidlá. Tie majú určitý formát v akom sú stavané, čo bude závisieť na tom aké výsledky poskytnú jednotlivé analýzy súborov.

Tvorba YARA pravidiel

Vytvoriť YARA pravidlo tak aby vyhovovalo gramatike jazyka YARA je pomerne jednoduché poskladaním správnej sekvencie reťazcov. Tento prístup však nie je vôbec ľahko udržovateľný, rozšíriteľný a veľmi náchylný na chyby. Ideálny spôsob je mať rozhranie, ktoré je

možné použiť zo zdrojového kódu, ktoré by sa staralo o vytvorenie pravidla. Knižnica *libyara*, ktorú ponúka samotný projekt YARA neprichádza s takouto možnosťou. Táto knižnica slúži len na preloženie už existujúceho pravidla do bajtkódu a následne jeho interpretáciu.

Navrhnutá knižnica `yaral` ponúka možnosť vytvárať YARA pravidlá pomocou programovateľného rozhrania. Na vstupe tohto rozhrania je možné nakonfigurovať ako chceme aby naše pravidlá vyzerali a na výstupe máme dostupný abstraktný syntaktický strom reprezentujúci celý súbor pravidiel. Nakoľko sa jedná o postranný výstup tejto práce, tak tu nijak dopodrobna nie je popísané, ako daná knižnica funguje.

Názvy .NET tried

Modul `dotnet` je dostupný až od verzie YARA 3.6 [5], ktorá počas implementovania tejto práce ešte nebola dostupná, ale ku dňu 23.5.2017 už je. Navyše tento modul ešte nepodporuje zisťovanie názvov tried. Stále je preto nutné riešiť .NET triedy pomocou obyčajných reťazcov. Experimentovanie ukázalo, že to dostatočné plní svoju činnosť.

Pre každú *skupinu pokrytia* sa vytvoria YARA reťazce `$dotnet_class_g<G>_<X>_p<P>`, kde `<G>` je poradové číslo skupiny, `<X>` je poradové číslo v rámci skupiny a `<P>` je percentuálne pokrytie danej skupiny. Praktický príklad je možný vidieť na ukážke 5.1.

Kód 5.1: Ukážková časť pravidla pre .NET triedy.

```
// ...
strings:
  $dotnet_class_g0_0_p100 = "Name1"
  $dotnet_class_g0_1_p100 = "Name2"
  $dotnet_class_g1_0_p090 = "Name3"
condition:
  ($dotnet_class_g0_0_p100 and $dotnet_class_g0_1_p100) or
      ↪ $dotnet_class_g1_0_p090
// ...
```

Názvy .NET metód

U .NET metód bude použitý podobný vzor ako u .NET. `dotnet_class` sa však zamení za `dotnet_method`.

TypeLib identifikátor

Jeden súbor môže obsahovať len jedno TypeLib identifikátor, takže nie je nutné dávať do názvu reťazca poradové číslo v rámci *skupiny pokrytia*, postačí len poradové číslo samotnej skupiny. Reťazce dostanú názov podľa vzoru `$type_lib_id_g<G>_p<P>`. Praktický príklad je možný vidieť na ukážke 5.2.

Kód 5.2: Ukážková časť pravidla pre TypeLib identifikátor.

```
// ...
strings:
  $type_lib_id_g0_p100 = "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
  $type_lib_id_g1_p100 = "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb"
condition:
  $type_lib_id_g0_p100 or $type_lib_id_g1_p100
// ...
```

Detekované reťazce

Pre detekované reťazce sa použije vzor $string_g<G>_<X>_p<P>. Navyše je špecifikovaný typ detekovaného reťazca. V prípade, že sa jedná o ASCII reťazec tak sa použije kvalifikátor ascii a v prípade *wide* reťazcov kvalifikátor wide. Praktický príklad je možný vidieť na ukážke 5.3.

Kód 5.3: Ukážková časť pravidla pre detakované reťazce.

```
// ...
strings:
  $string_g0_0_p100 = "AAA"
  $string_g0_1_p100 = "BBB" wide
condition:
  $string_g0_0_p100 and $string_g0_1_p100
// ...
```

Importované symboly

Importované symboly nevyžadujú tvorbu žiadneho YARA reťazca, pretože je možné využiť modul pe a vytvoriť podmienku za použitia funkcie imports. Praktický príklad je možný vidieť na ukážke 5.4.

Kód 5.4: Ukážková časť pravidla pre importované symboly.

```
// ...
condition:
  pe.imports("KERNEL32.DLL", "ExitProcess") and pe.imports("WS2_32.DLL", 1)
// ...
```

Vstupný bod

U vstupného bodu sa využívajú hexadecimálne reťazce na definíciu bajtov, ktorých názov je daný podľa vzoru $entry_point_g<G>_p<P>. Do výslednej podmienky je zostrojený výraz spájajúci adresu a bajty na vstupnom bode, výsledná časť YARA pravidla je potom zobrazená na ukážke 5.5.

Kód 5.5: Ukážková časť pravidla pre vstupný bod.

```
// ...
strings:
  $entry_point_g0_p100 = { A1 B2 C3 D4 E5 F6 }
condition:
  pe.entry_point == 0x1000 and $entry_point_g0_p100 at pe.entry_point
// ...
```

Sekcie

Pre sekcie je možné priamo využiť module pe, konkrétne zoznam sprístupňujúci sekcie a ich atribúty. Term výslednej podmienky vyzerá podľa ukážky 5.6.

Kód 5.6: Ukážková časť pravidla pre sekcie.

```
// ...
condition:
  pe.sections[0].name == ".text" and pe.sections[1].name == ".data"
// ...
```

Cesta k PDB súboru

Modul pe nesprístupňuje cestu k PDB súboru zapísanom v *debug directory* a preto je opäť nutné použiť rovnakú techniku ako u TypeLib identifikátora a ísť cez obyčajné reťazce. Vzor pre názov reťazca je `$pdb_g<G>_p<P>`.

Rich hlavička

Ako bolo spomenuté v kapitole 2.6, *rich* hlavička sa nachádza v PE súbore zašifrovaná pomocou 4 bajtového kľúča. YARA sprístupňuje *rich* hlavičku ako v zašifrovanej, tak aj odšifrovanej podobe. Nakoľko je prešifrovanie *rich* hlavičky triviálna činnosť, tak je lepšie ak spravíme podmienku z odšifrovanej podoby. Vznikne formula, ktorá je na ukážke 5.7.

Kód 5.7: Ukážková časť pravidla pre *rich* hlavičku.

```
// ...
condition:
  pe.rich_signature.clear_data == "DanS\x00\x01AB"
// ...
```

Podpisujúci

Aj v tomto prípade pomôže modul pe umožňujúci vytvoriť podmienku priamo na certifikát podpisujúceho. Konkrétne sa zameriame na celkovú identifikáciu subjektu. Ukážka 5.8 zobrazuje vygenerovanú formulu.

Kód 5.8: Ukážková časť pravidla pre podpisovateľa.

```
// ...
condition:
  pe.signatures[0].signer == "C=CZ/L=Brno/CN=Milkovic/O=FIT"
// ...
```

Formule YARA podmienky

Každá analýza, ktorá úspešne dobehne do konca a vyprodukuje nejaké *skupiny pokrytia* spôsobí vygenerovanie formule, ktorá sa objaví vo výslednom YARA pravidle. Každá hodnota vygeneruje buďto term, alebo formulu. V rámci jednej *skupiny pokrytia* sú všetky termy (resp. formule) spojené konjunkciou. V prípade, že v *skupine pokrytia* sa nachádzajú viac ako 4 hodnoty a je to pre daný term (resp. formulu) možné, tak sa použije skrátená varianta pomocou výrazu `all of (...)`, kde `...` predstavuje zoznam reťazcov.

Výstupný detekčný vzor

Výstupný detekčný vzor, vo forme YARA súboru začína metadátami v komentároch. Ich charakter je čisto informatívny. Tieto komentáre obsahujú počet vygenerovaných verejných pravidiel (zatiaľ je toto číslo stále 1) a počet súborov, z ktorých bol detekčný vzor vytvorený.

Ďalej sa súbor skladá z jedného pravidla. Toto pravidlo dostane buď meno podľa konfigurácie vo forme `rule_<NAME>_static`. Ak táto možnosť v konfigurácií chýba, tak je meno dané podľa aktuálneho dátumu a času vo formáte `rule_%Y%m%d_%H%M%S_static`[2]. Metadáta pravidla obsahujú vždy nasledovné hodnoty:

[2]Formátovanie dátumu je dané štandardnou C/C++ funkciou `strftime`.

- `author` – Ak nie je definovaný z konfigurácie autor pravidla, tak obsahuje reťazec `yaragen` nasledovaný verziou nástroja. Pokiaľ je v konfigurácií uvedený autor, tak sa táto informácia nachádza v zátvorka za autorom z konfigurácie.
- `description` – Slovný popis pravidla. Štandardne prázdne.
- `reliability` – Spoľahlivosť pravidla. Obsahuje interné hodnoty AVG Technologies pre klasifikáciu. Štandardne prázdne.
- `strain` – Rodina *malwaru*, ktorej detekciu zachycuje toto pravidlo. Štandardne prázdne.
- `type` – Typ *malwaru*. Obsahuje hodnoty ako `adware`, `ransomware`, `trojan` atď. Štandardne prázdne. Táto hodnota sa ďalej používa ku klasifikácií *malwaru* automatizovanými nástrojmi spoločnosti AVG Technologies.

Podmienku pravidla je zložená z formúl definovaných jednotlivými analýzami, ktoré boli spomenuté vyššie. Jednotlivé formuly analýz sú spájané logickým operátorom disjunkcie. Disjunkcia bola zvolená z dôvodu, že vygenerovaný detekčný vzor nie je určený pre antivírusové jadro a nebude sa starať o detekciu v klientských počítačoch. Tento detekčný vzor je určený pre analytikov, ktorý budú dané vzorky dodatočne analyzovať a preto je vhodné, ak výsledné YARA pravidlo bude mať väčší záber a analytik vyradí niektoré vzorky ako chyby typu 1 (*false positive*), ako keby sa k analytikovi dostala menšia sada nepokrývajúca aj mierne odlišné vzorky.

Kód 5.9: Ukážka detekčného vzoru bez restriktívnych analýz.

```
// Rules: 1
// Covered files: 2854

import "pe"

rule rule_20170515_010604_static {
  meta:
    author = "yaragen␣0.2"
    description = ""
    reliability = ""
    strain = ""
    type = ""
  strings:
    $pdb_g0_p100 = "E:\\svn\\driver_reconstitution\\avdriver_proj\\Driver\\
        ↪ bd0001_x64\\src\\dll\\objfre64\\amd64\\bd0001.pdb"
    $entry_point_g0_p100 = { B8 01 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
        ↪  48 83 EC 58 48 89 5C 24 60 48 89 6C 24 68 48 89 74 24 70 48 89
        ↪ 7C 24 78 49 8B F0 8B EA 49 8B F9 48 8B }
  condition:
    ($pdb_g0_p100) or
    (pe.rich_signature.clear_data == "DanS\x00\x00\x00\x00\x00\x00\x00\x00\
        ↪ x00\x00\x00\x00v\x9dz\x00\x01\x00\x00\x00v\x9d}\x00\x02\x00\x00\
        ↪ x00v\x9dm\x00\x06\x00\x00\x00\xc3\x0f^\x00\x01\x00\x00\x00v\x9dx\
        ↪ x00\x01\x00\x00\x00") or
    ((pe.entry_point == 0x40002a80 and $entry_point_g0_p100 at
        ↪ pe.entry_point))
}
```

Je však možnosť pre analytika ako toto chovanie zmeniť pomocou konfigurácie analýz. V podkapitole 5.2.2 bola spomenutá restriktívnosť analýz, ktorá je štandardne vypnutá. Pokiaľ sa však v konfigurácií označí analýza ako restriktívna, tak to bude mať na generátor vplyv, že medzi restriktívne analýza sa spoja konjunkciou a nerestriktívne disjunkciou. Restriktívna a nerestriktívna formula sa spoja konjunkciou. Týmto spôsobom je možnosť vynútiť užší záber YARA pravidla. Takéto pravidlo s restriktívnou analýzou pre cestu k PDB súboru a *rich* hlavičku je zobrazené na ukážke 5.10.

Kód 5.10: Ukážka detekčného vzoru s restriktívnymi analýzami pre PDB a *rich* hlavičku.

```
// Rules: 1
// Covered files: 2854

import "pe"

rule rule_20170515_010831_static {
  meta:
    author = "yaragen␣0.2"
    description = ""
    reliability = ""
    strain = ""
    type = ""
  strings:
    $pdb_g0_p100 = "E:\\svn\\driver_reconstitution\\avdriver_proj\\Driver\\
        ↪ bd0001_x64\\src\\dll\\objfre64\\amd64\\bd0001.pdb"
    $entry_point_g0_p100 = { B8 01 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
        ↪  48 83 EC 58 48 89 5C 24 60 48 89 6C 24 68 48 89 74 24 70 48 89
        ↪ 7C 24 78 49 8B F0 8B EA 49 8B F9 48 8B }
  condition:
    ($pdb_g0_p100) and
    (pe.rich_signature.clear_data == "DanS\x00\x00\x00\x00\x00\x00\x00\x00\
        ↪ x00\x00\x00\x00v\x9dz\x00\x01\x00\x00\x00v\x9d}\x00\x02\x00\x00\
        ↪ x00v\x9dm\x00\x06\x00\x00\x00\xc3\x0f^\x00\x01\x00\x00\x00v\x9dx\
        ↪ x00\x01\x00\x00\x00") and (
      ((pe.entry_point == 0x40002a80 and $entry_point_g0_p100 at
          ↪ pe.entry_point))
    )
}
```

Špeciálnym prípadom je pokiaľ sú všetky vstupné súbory vyhodnotené ako .NET. V tomto prípade chceme vymedziť aby aj detekčný vzor bol definovaný výhradne pre .NET súbory. Experimentami nad 15000 .NET súbormi sa ukázalo, že najväčšiu úspešnosť má pravidlo zobrazené na ukážke 5.11. Odkaz na toto pravidlo sa doplní do výslednej YARA podmienky, ako keby sa jednalo o restriktívnu analýzu.

Kód 5.11: Pravidlo vymedzujúce .NET súbory.

```
private rule MSIL {
  strings:
    $mscoree = "mscoree.dll" ascii wide nocase
  condition:
    $mscoree
}
```

5.2.4 Konfigurácia

Konfiguráciu je možno poskytnúť skrz konfiguračný súbor vo formáte INI. Konfigurovateľné možnosti môžeme rozdeliť na globálne, meta hodnoty a tie, ktoré patria ku konkrétnej analýze. Tie globálne sa nachádzajú na najvyššej úrovni v INI súbore. Medzi ne patrí:

- `analyses` – CSV (angl. *Comma Separated Values* - hodnoty oddelené čiarkami) zoznam identifikátorov analýz, ktoré sa majú vykonať.
- `exclude` – CSV zoznam identifikátorov analýz, ktoré sa majú vynechať. Len jedna z možností `analyses` a `exclude` je možnú použiť. V prípade, že sú špecifikované obe, tak `analyses` je prioritizovaná.
- `rule-name` – Názov pravidla, ktorý sa má použiť namiesto časovej značky.

Ďalšou možnosťou, čo konfigurovať sú vlastné meta informácie do výstupného YARA pravidla. Špecifikovať ich musíme vo vlastnej INI sekcií `[meta]`. Každý pár kľúč–hodnota v tejto sekcií je odzrkadlený ako kombinácia kľúč–hodnota v meta informáciach pravidla.

Výstupný súbor je taktiež konfigurovateľný sekciou `output`. Buďto je možné špecifikovať ako `output.static` pre statický detekčný vzor, alebo `output.all`, ktorý má zatiaľ rovnaké chovanie ako `output.static` a slúži na budúce použitie.

Konfigurácia analýz sa špecifikuje v sekciách s identifikátorom analýzy. Každá analýza má niekoľko možných konfigurovateľných hodnôt, ktoré boli popísané v podkapitole 5.2.2. Identifikátory jednotlivých analýz a k nim možné konfiguračné atribúty sú:

- Spoločné možnosti pre všetky analýzy.
  - `min_percent_coverage` – Minimálne percentuálne pokrytie *skupiny pokrytia*. Hodnota 0 až 100.
  - `must_cover_all` – Povinnosť pokrytia všetkých vstupných súborov. Hodnota 0 alebo 1.
  - `restrictive` – Restriktívnosť analýzy. Hodnota 0 alebo 1.
- `dotnet_class` – Analýza názvov .NET tried.
  - `min_group_size` – Minimálna veľkosť *skupiny pokrytia*.
  - `min_name_length` – Minimálna dĺžka názvu .NET triedy.
- `dotnet_method` – Analýza názvov .NET metód.
  - `min_group_size` – Minimálna veľkosť *skupiny pokrytia*.
  - `min_name_length` – Minimálna dĺžka názvu .NET metódy.
- `type_lib_id` – Analýza TypeLib identifikátora.
- `strings` – Analýza detekovaných reťazcov.
  - `min_group_size` – Minimálna veľkosť *skupiny pokrytia*.
  - `min_length` – Minimálna dĺžka reťazca.
- `imports` – Analýza importovaných symbolov.
  - `min_group_size` – Minimálna veľkosť *skupiny pokrytia*.

- `entry_point` – Analýza vstupného bodu.
  - `bytes_count` – Počet bajtov na vstupnom bode.
- `sections` – Analýza sekcií.
  - `min_group_size` – Minimálna veľkosť *skupiny pokrytia*.
- `pdb` – Analýza cesty k PDB súboru.
- `rich_header` – Analýza *rich* hlavičky.
- `signer` – Analýza podpisujúceho.

Kapitola 6

Implementácia nových analýz a generovania detekčného vzoru

Obsahom tejto kapitoly je implementácia navrhnutých analýz do knižnice *fileformatl* a nástroja na generovanie detekčných vzorov, popísaných v rámci kapitoly 5.

6.1 Implementácia nových analýz

Knižnica *fileformatl*, ktorá bola popísaná v podkapitole 4.2, je implementovaná v jazyku C++ podľa štandardu ISO C++14 [10], preto aj na implementáciu nových analýz bude použitý práve tento jazyk. Pre každú analýzu je vždy doplnené parsovanie dát zo súboru a ich prevod do internej reprezentácie.

V nasledujúcich podkapitolách je popísaný spôsob implementácie jednotlivých analýz navrhnutých v podkapitole 5.1.

6.1.1 Certifikáty a podpis

Knižnica *fileformatl* využíva na spracovanie formátu PE už vyše 10 rokov nevyvíjanú knižnicu *pelib* [13], ktorú však interne rozširujeme a udržujeme v rámci projektu RetDec [14]. Táto knižnica ale neobsahuje žiadnu podporu pre *security directory*. Prvým krokom je tým pádom implementácia parsovania *security directory* v knižnici *pelib*.

Knižnica pelib

Do súboru `PeLibAux.h` sú doplnené nové potrebné konštanty súvisejúce s verziou *security directory* a jeho typu obsahu. Do rovnakého súboru je tiež pridaná definícia štruktúry samotného *directory*.

Novovytvorené súbory `SecurityDirectory.cpp` a `SecurityDirectory.h` obsahujú definíciu resp. deklaráciu triedy `SecurityDirectory`. Ako je zvykom v rozhraní tried knižnice *pelib*, tak táto trieda obsahuje metódu `SecurityDirectory::read`, ktorá načíta obsah odpovedajúceho *data directory* zo súboru a vráti buď chybu, alebo uspeje a vráti `ERROR_NONE`. `Certificate` je naplnený binárnym obsahom PKCS7 štruktúry.

Tento nový *data directory* je ale potrebné aj nejak sprístupniť z rozhrania triedy `PeFile`, ktorá predstavuje vstupné rozhranie knižnice *pelib*. Nová metóda `PeFile::securityDir` vráti objekt triedy `SecurityDirectory`. Jednotlivé *data directories* sa načítavajú v knižnici

pelib až po explicitnom vyžiadaní metódou PeFile::readSecurityDirectory, ktorá len zavolá ďalej metódu SecurityDirectory::read so správnou adresou.

Knižnica fileformatl

V rámci *fileformatl* sa budeme zaoberať primárne triedou PeFormat, ktorá tvorí abstrakciu nad súbormi vo formáte PE. Je potrebné však abstrahovať aj samotné certifikáty, na čo slúžia nové triedy CertificateTable a Certificate medzi ktorými je vzťah, že jedna CertificateTable môže obsahovať niekoľko objektov typu Certificate. Atribúty triedy Certificate odpovedajú návrhu výstupu z podkapitoly 5.1.1. Trieda CertificateTable udržiava certifikáty v STL (angl. *Standard Template Library*) kontajnery std::vector. Naviac obsahuje index certifikátu podpisovateľa a protipodpisovateľa.

Parsovanie štruktúry PKCS7 sa deje v metóde PeFormat::loadCertificates. Nerobí sa to však manuálne, ale je použité existujúce riešenie v knižnici OpenSSL [12] cez funkciu d2i_PKCS7_bio, ktorá DER kódovanie PKCS7 štruktúry zapísané v BIO[1] vstupe transformuje do internej OpenSSL reprezentácie a tú vráti. Certifikát podpisovateľa sa získa funkciou PKCS7_get0_signers, pričom sa zvolí index 0, pretože súbory PE budú mať vždy iba jedného podpisovateľa. Zoznam všetkých certifikátov sa zase získa prístup k atribútu d.sign->cert. Prítomnosť protipodpisu sa zisťuje pomocou prítomnosti atribútu NID_pkcs9_countersignature v štruktúre, ktorú vráti PKCS7_get_signer_info. Tento atribút však obsahuje len sériové číslo a názov vystavovateľa certifikátu. Certifikát protipodpisovateľa sa preto následne získa prehľadaním všetkých certifikátov pomocou funkcie X509_find_by_issuer_and_serial, ktorá vyhľadá konkrétny certifikát na základe špecifikovaného sériového čísla a názvu vystavovateľa certifikátu.

Posledným krokom pri parsovaní je rekonštrukcia certifikačného reťazca podpisu a protipodpisu. Ten sa zostrojí ako je to popísané v podkapitole 5.1.1. Pri zostrojovaní tohto reťazca sa vytvárajú objekty typu Certificate a ukladajú sa do CertificateTable. Pri tvorbe objektov Certificate sa spúšťa metóda Certificate::load, ktorá parsuje samotnú X509 štruktúru reprezentujúcu certifikát v OpenSSL forme. Jednotlivé atribúty certifikátu sú parsované nasledovne:

- Platnosť od-do – Dátum, ktorý vráti X509_get_notBefore a X509_get_notAfter je pomocou ASN1_TIME_print prevedený na reťazec.
- Verejný kľúč – Získava sa pomocou funkcie X509_get_pubkey. Parsovaný je v podobe hexadecimálneho reťazca z PEM formátu PEM_write_bio_PUBKEY. Šifrovací algoritmus sa získa ako reťazec pomocou i2a_ASN1_OBJECT.
- Algoritmus podpisu – Tak isto ako šifrovací algoritmus, cez funkciu i2a_ASN1_OBJECT.
- Sériové číslo – Prevedie sa do hexadecimálnej podoby a uloží ako reťazec. Získané cez X509_get_serialNumber.
- Podpisovateľ a vystavovateľ – Identifikácia sa získa ako *one-line* reťazec pomocou X509_NAME_oneline zavolanej na výsledok X509_get_subject_name pre podpisovateľa a X509_get_issuer_name pre vystavovateľa.

[1] Jedná sa o mechanizmus abstrakcie vstupu/výstupu používaný naprieč celou knižnicou OpenSSL umožňujúci reťazenie.

Overenie podpisu sa taktiež rieši v rámci `PeFormat::loadCertificates` odkiaľ sa volá metóda `PeFormat::verifySignature`. Ako je popísané v podkapitole 5.1.1, tak overenie podpisu sa skladá z porovnania hashu obsahu a následne verifikácia verejným kľúčom. Hash voči ktorému porovnávame vypočítaný hash súboru sa nachádza v `contentInfo` časti PKCS7 štruktúry. OpenSSL však ponúka tento atribút iba vo forme binárnych dát, ktoré si musíme naparsovať sami. Preto bol vytvorený vlastný ASN.1 parser DER kódovania.

Základom ASN.1 parseru je trieda `Asn1Item` predstavujúca abstraktný typ. Z nej dedí mnoho ďalších tried reprezentujúcich konkrétne typy a to:

- Žiadna hodnota – `Asn1Null`
- Bitový reťazec – `Asn1BitString`
- Bajtový reťazec – `Asn1OctetString`
- Objekt – `Asn1Object`
- Sekvencia – `Asn1Sequence`
- Kontextovo závislá hodnota – `Asn1ContextSpecific`

Trieda `Asn1Item` obsahuje statickú metódu `Asn1Item::parse`, ktorá z binárnych dát vytvorí objekt správneho typu a vráti ukazovateľ na neho.

Z atribútu `contentInfo` sa získa referenčný hash a algoritmus hashu podľa popisu obsahu v podkapitole 2.3 pomocou ASN.1 parseru. Intervaly, ktoré je nutné zahashovať sa vypočítajú nasledovne:

- Od začiatku súboru po `m_checksumFileOffset`. Preskakujú sa nasledujúce 4 bajty.
- Od `m_checksumFileOffset + 4` až po `m_secDirFileOffset`
- Od `m_secDirFileOffset + 8` až po pozíciu *security directory* v súbore
- Od konca *security directory* po koniec súboru.

Vypočítaný hash sa porovná a pokiaľ sedia, tak sa zavolá funkcia `PKCS7_verify`, ktorá verifikuje podpis.

6.1.2 Rekonštrukcia .NET typov

Implementácia rekonštrukcie .NET typov je rozdelená do dvoch častí. Najskôr je totiž nutné z knižnice *pelib* získať informácie o CLR hlavičke, metadata hlavičke a metadata *streamoch*. V druhej časti sa potom použijú tieto informácie na rekonštrukciu samotných typov.

Načítavanie .NET hlavičiek začína v metóde `PeFormat::loadDotnetHeaders` v súbore `pe_format.cpp`. Najskôr sa prečíta CLR hlavička do objektu `clrHeader`. Prepočíta sa relatívna adresa na virtuálnu, aby sa zistilo umiestnenie *metadata* hlavičky. Zistí sa prítomnosť jednotlivých *streamov*, ktoré sa načítajú do odpovedajúcich tried.

Metadata stream je spracovaný v metóde `parseMetadataStream`, ktorá na základe 64 bitovej masky zistí, ktoré *metadata* tabuľky sú prítomné a pomocou šablónovej metódy `parseMetadataTable` načíta patričnú tabuľku. Tie sú implementované v podobne šablónovej triedy `MetadataTable<T>`, kde šablónovým typom je typ záznamu v tabuľke. Základným typom jedného záznamu je abstraktná trieda `BaseRecord`. Abstraktná metóda `load`

prijíma ako parametre instanciu triedy `FileFormat`, ukazovateľ na `MetadataStream` a referenciu na adresu, z ktorej sa má začať načítavať záznam do tabuľky. Tento parameter je zároveň výstupný a pri opustení danej metódy obsahuje adresu nasledujúceho záznamu. Z triedy `BaseRecord` potom dedia ostatné triedy, ktoré predstavujú už konkrétne záznamy jednotlivých tabuliek.

Druhá fáza začína detekciou TypeLib identifikátora a rekonštrukciou samotných dátových typov. Triedy na reprezentáciu jednotlivých .NET typov sú:

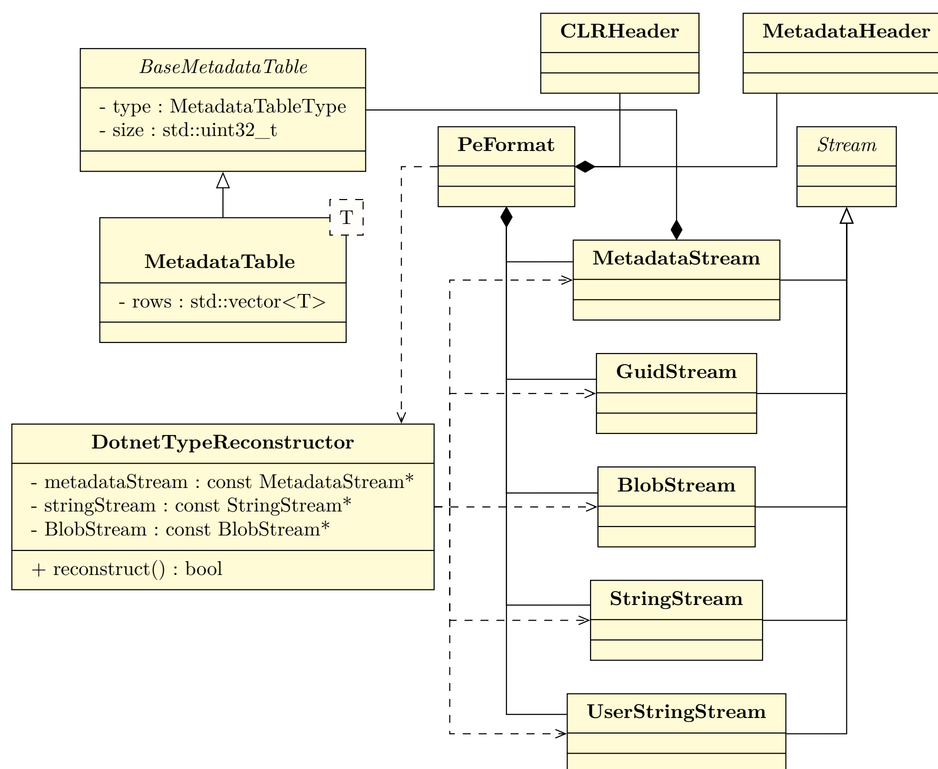
- `DotnetClass` – trieda
- `DotnetMethod` – metóda
- `DotnetProperty` – vlastnosť
- `DotnetField` – atribút
- `DotnetParameter` – parameter metódy

Každá z týchto tried dedí zo základnej triedy `DotnetType`. Dátový typ je reprezentovaný vo forme triedy `DotnetDataTypeBase`. Šablónová trieda `DotnetDataType` s parametrom `Type` typu `ElementType` potom reprezentuje dátový typ elementárneho typu `Type`. Špecializáciou tejto šablóny pre zložené elementárne typy je tvorená hierarchia dátových typov. Medzi Zložené elementárne typy sú:

- `ElementType::Ptr` – ukazovateľ.
  - Ukazovaný typ (`DotnetDataTypeBase`).
- `ElementType::ByRef` – referencia.
  - Odkazovaný typ (`DotnetDataTypeBase`).
- `ElementType::ValueType` – odkaz na triedu.
  - Typ triedy (`DotnetClass`).
- `ElementType::Class` – odkaz na triedu (nerozlíšiteľné od `ValueType`)
  - Typ triedy (`DotnetClass`).
- `ElementType::GenericVar` – generický parameter u tried.
  - Názov generického parametra (`std::string`).
- `ElementType::Array` – pole.
  - Typ prvku pola (`DotnetDataTypeBase`).
  - Rozmery jednotlivých dimenzií poľa.
- `ElementType::GenericInst` – instancia konkrétneho generického typu.
  - Generický typ (`DotnetDataTypeBase`).
  - Generické parametre (`DotnetDataTypeBase`).
- `ElementType::FnPtr` – ukazovateľ na funkciu.

  - Návratový typ (`DotnetDataTypeBase`).
  - Typ parametrov ako zoznam (`DotnetDataTypeBase`).
- `ElementType::SzArray` – jednorozmerné pole bez definovanej veľkosti.
  - Typ prvku pola (`DotnetDataTypeBase`).
- `ElementType::GenericMVar` – generický parameter u metód.
  - Názov generického parametra (`std::string`).
- `ElementType::CModRequired` – špeciálny modifikátor typu.
  - Modifikátor (`DotnetClass`).
  - Modifikovaný typ (`DotnetDataTypeBase`)

Rekonštrukcia sa deje v triede `DotnetTypeReconstructor`.



Obr. 6.1: Zjednodušený diagram tried pre rekonštrukciu .NET typov.

6.1.3 Detekcia reťazcov

Detekcia reťazcov je implementovaná na úrovni triedy `FileFormat` v knižnici *fileformatl*. Do nej je pridaných niekoľko preťažených variant metódy `loadStrings`. Prvá varianta, bez parametrov, načíta ako ASCII tak aj *wide* reťazce pomocou varianty `loadStrings` prijímajúcej typ reťazca, ktoré načítavať a akú veľkosť v bajtoch má 1 znak. Následne zoradí tento zoznam lexikograficky. Spomínaná druhá varianta `loadStrings` preitereruje

cez všetky sekcie v súbore a zavolá tretiu variantu `loadString`, ktorá navyše oproti tej druhej očakáva aj parameter sekcie, z ktorej sa majú reťazce načítavať.

Pre minimalizovanie kopírovania dát z jedného miesta v pamäti na druhé bol vytvorený špeciálny druh iterátoru nazvaný `CharacterIterator` čiastočne splňujúci koncept `BidirectionalIterator`. Tento iterátor je schopný prechádzať cez ľubovoľnú iterovateľnú sekvenciu, splňujúcu koncept `RandomAccessIterator`, po krokoch rôznej dĺžky s možnosťou kontroly, či sa práva nachádza na valídnom znaku alebo nie. Zároveň podporuje bezpečnú kontrolu medzí, aby nebolo možné prekročiť iterátorom za koniec postupnosti. Aby tiež bolo možné použiť tento vlastný iterátor vo volaniach funkcií zo štandardnej knižnice, tak je vytvorená špecializácia `std::iterator_traits` štruktúry poskytujúca niektoré nutné typy, ako `iterator_category`, `value_type`, či `difference_type`. Nakoľko dedukcia typov v šablónach pri inicializácií funguje až od C++17, tak je zároveň dostupná funkcia `makeCharacterIterator` u ktorej dedukcia funguje. V prípade, že sa pomocou tohto iterátoru narazí na sekvenciu platných znakov dlhú aspoň `DefaultMinStringLength`, tak sa pridá tento reťazec k detekovaným.

Nakoľko je volanie konštruktorov definované v poradí od najvyššej triedy v hierarchií až po tú najnižšie, tak je nutné doplniť volanie `loadStrings` až do inicializácie špecifických tried jednotlivých formátov (`PeFormat`, `ElfFormat`, ...), tak aby v momente keď je táto metóda zavolaná, tak už boli načítané všetky sekcie. To je napríklad na koniec metódy `init`, ktorá je volaná z konštruktora týchto podtried.

6.2 Generovanie detekčného vzoru

Nástroj, ktorý dostal názov *yaragen* (z názvov YARA a generátor), je napísaný v jazyku C++ podľa štandardu ISO C++14 [10]. Externými závislosťami sú knižnica *boost* vo verzií 1.64.0 [3] a časť rekonfigurovateľného spätného prekladača RetDec [14]. Z dôvodu jednoduchosti použitia a integrácie tohto nástroja do ľubovolného systému je nástroj rozdelený na dve hlavné časti:

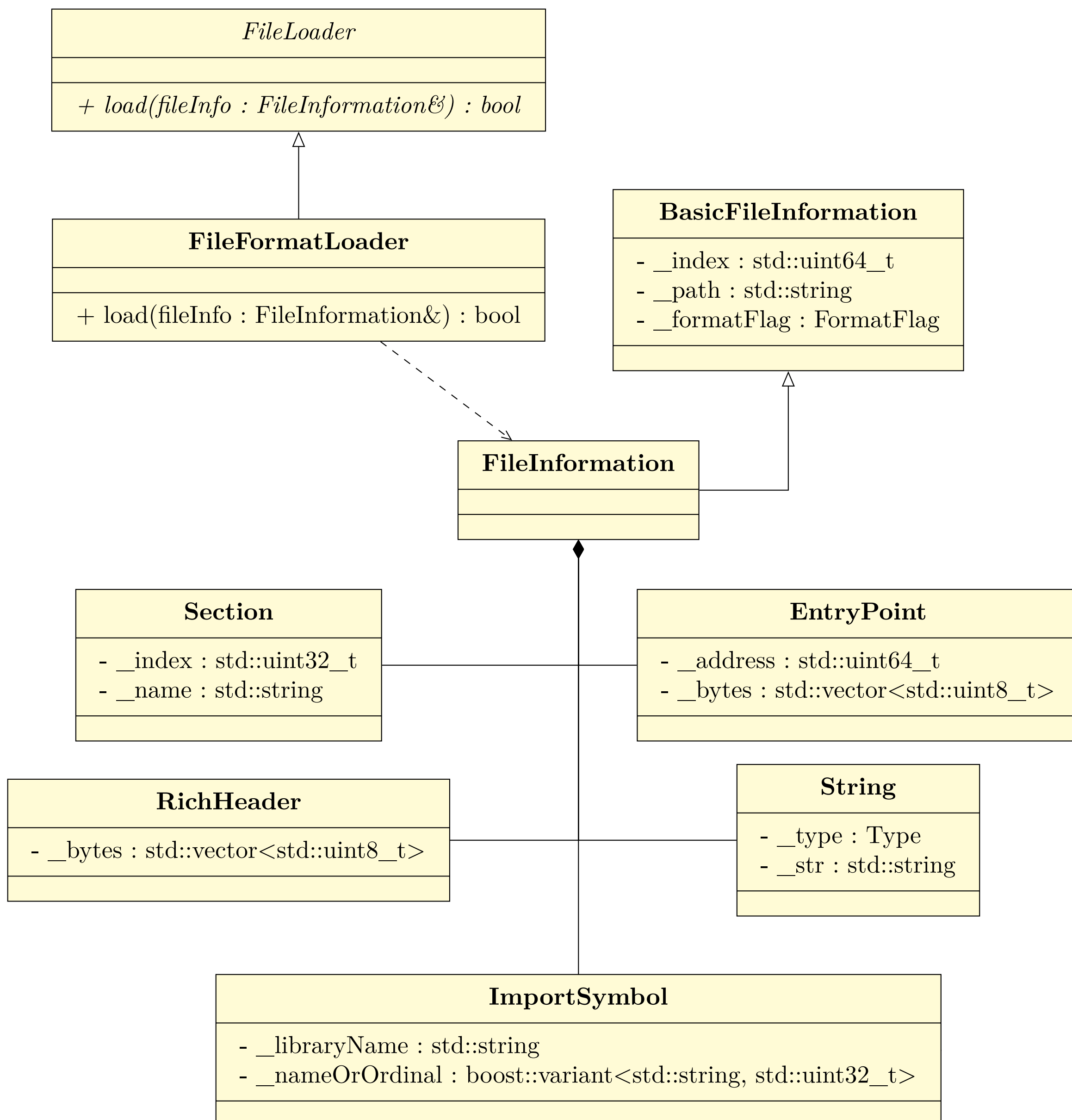
- Knižnica `yaragenlib`, ktorá implementuje celkovú funkčnosť. To zahrňuje extraktor, analyzátor a generátor.
- Nástroj `yaragen`, poskytujúci rozhranie knižnice pomocou konzolového nástroja.

V nasledujúcich podkapitolách sú popísané implementácie jednotlivých modulov, ako boli navrhnuté v kapitole 5.2.

6.2.1 Extraktor

Základné informácie o vstupnom súbore sa nachádzajú v triede `BasicFileInformation`. Konkrétne sa jedná o index spracovávaného súboru, cesta k súboru a jeho formát. Táto trieda je rozšírená pomocou podtriedy `FileInformation` o extrahované informácie použité pri analýze. Extraktor samotný je implementovaný vo forme abstraktnej triedy `FileLoader`, ktorá obsahuje abstraktnú metódu `load(FileInformation&)`. Potomkovia musia poskytnúť implementáciu tejto metódy a doplniť do poskytnutého objektu cez parameter vyextrahované informácie.

Extraktor využívajúci knižnicu *fileformatl* je implementovaný vo `FileFormatLoader` triede. Jeho metóda `load` vytvorí instanciu objektu `FileFormat` a pomocou jej rozhrania vyextrahuje potrebné informácie. Zároveň tu dochádza k rozpoznaniu .NETu od obyčajného PE.



Obr. 6.2: Zjednodušený diagram tried pre extraktor.

6.2.2 Analyzátor

Základ analyzátora tvorí trieda `AnalysisEngine`, ktorá spúšťa jednotlivé analýzy. Metóda `run` prijíma ako parameter ukazovateľ na `FileLoader`, aby vedel, ktorý extraktor používať. Prvým krokom je načítanie ciest pre analyzované súbory. Poskytnuté vstupné cesty totiž môžu byť aj zložky, ktoré je preto nutné rekurzívne prejsť a zistiť prítomné súbory. Na to sa použije modul `boost::filesystem`, ktorý nahradzuje zatiaľ chýbajúcu podporu pre unifikovanú prácu so súborovým systémov v štandarde jazyka C++. Každá analýza je najskôr spustená pomocou šablónovej metódy `runAnalysis` a po tom čo sú zanalyzované všetky súbory, tak sa analýzy finalizujú šablónovou metódou `finalizeAnalysis`.

Základom pre tvorbu jednotlivých analýz je abstraktná trieda `BaseAnalysis`. Obsahuje jedine identifikáciu konkrétnej analýzy, jej popis a formáty, pre ktoré je určená. Táto trieda existuje len z dôvodu aby bolo možné každú analýzu jednotne uložiť do homogénneho zoznamu pod ukazovateľom rovnakého typu. Z tejto trieda je dedená ďalšia abstraktná trieda `Analysis`. Jedná sa o šablónovú triedu, ktorá očakáva nasledovné šablónové parametre:

- `AnalysisT` – Pri dedení tejto triedy je nutné do tohto šablónového parametru dať typ, ktorý dedíme, tj. samotnú analýzu, ktorú vytvárame. Táto technika je známa ako CRTP (*Curiously Recurring Template Pattern*) [38]. V tomto prípade je tu z dôvodu vynútenia existencie statických premenných `info` a `defaultSettings`.
- `ResultT` – Typ, ktorý je použitý pre výsledok analýzy. Musí sa jednať o potomka triedy `Analysis`. Ďalej v texte bude spomenuté viac o tejto triede.
- `SettingsT` – Dátový typ nastavení. Musí byť potomok triedy `AnalysisSettings`, ktorá je popísaná ďalej v texte.

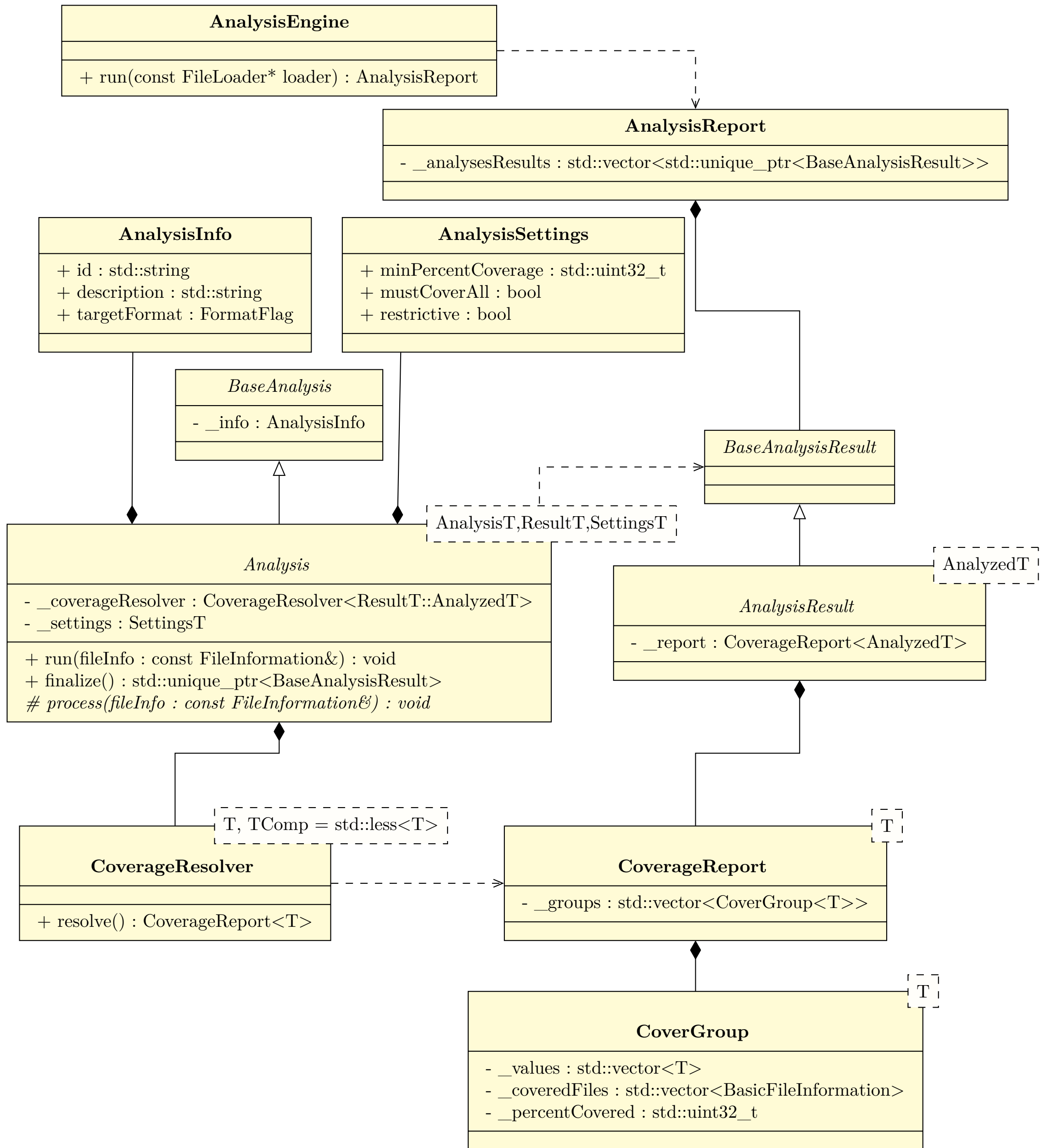
Ako bolo spomenuté, každá analýza vracia svoj výsledok, ktorý má dátový typ podľa šablónového parametra. Pre tieto výsledky tiež existuje hierarchia tried. Základom je trieda `BaseAnalysisResult`, ktorá slúži na rovnaké účely ako `BaseAnalysis`. Z nej dedí trieda `AnalysisResult` majúca šablónový parameter `AnalyzedT`, ktorý predstavuje typ analyzovanej hodnoty. Jednotlivé triedy výsledkov konkrétnych analýz sú zároveň zaradené do návrhového vzoru *visitor* ako navštevovaný typ. Každá trieda dediaca z `AnalysisResult` je povinná implementovať metódu `accept` v tvare ako je to zobrazené na ukážke 6.1. Pre vytvorenie návštevníka postačí dediť z `AnalysisResultVisitor` a poskytnúť implementácie metódam `visit`.

Kód 6.1: Ukážka povinnej implementácie metódy `accept`.

```
void RESULT_TYPE::accept(AnalysisResultVisitor* visitor) const
{
    visitor->visit(this);
}
```

Pokrytie súborov analyzovanými hodnotami rieši `CoverageResolver`. Jeho instancia je dostupná v triede `Analysis` pod názvom `_coverageResolver`. Metódami `addPreFilter` a `addPostFilter` je možné doňho pridať *pre* a *post* filtre. Parametrom je ľubovolný volateľný objekt, či už je to lambda funkcia, instancia `std::function` alebo ukazovateľ na funkciu. Registrácia súboru sa vykonáva pomocou metódy `registerFile` a jednotlivé hodnoty sa pridávajú metódou `addValueForFile`. Metóda `resolve` vypočíta všetky *skupiny pokrytia*. Pri ich počítaní je nutné porovnať rovnosť množín súborov. Pre jednoduché porovnanie dvoch usporiadaných množín sa vytvorí `std::unordered_map`, indexovaná usporiadaným `std::vector`-om. K1účová hodnota v `std::unordered_map` musí byť hashovateľná a na to sa použije rýchla nekryptografická hashovacia funkcia FNV1a [6]. Výsledkom metódy `resolve` je objekt typu `CoverageReport` obsahujúci *skupiny pokrytia* vo forme objektov `CoverGroup`.

Výsledky všetkých analýz sú nakoniec uložené do objektu typu `AnalysisReport`. Táto trieda zároveň obsahuje metódy na zistenie toho, či všetky vstupné súbory boli vo formáte .NET pomocou `allDotnet`, alebo ktoré všetky súbory sú pokryté cez `getAllCoveredFiles`. Tento objekt je ďalej posunutý generátoru.



Obr. 6.3: Zjednodušený diagram tried pre analyzátor.

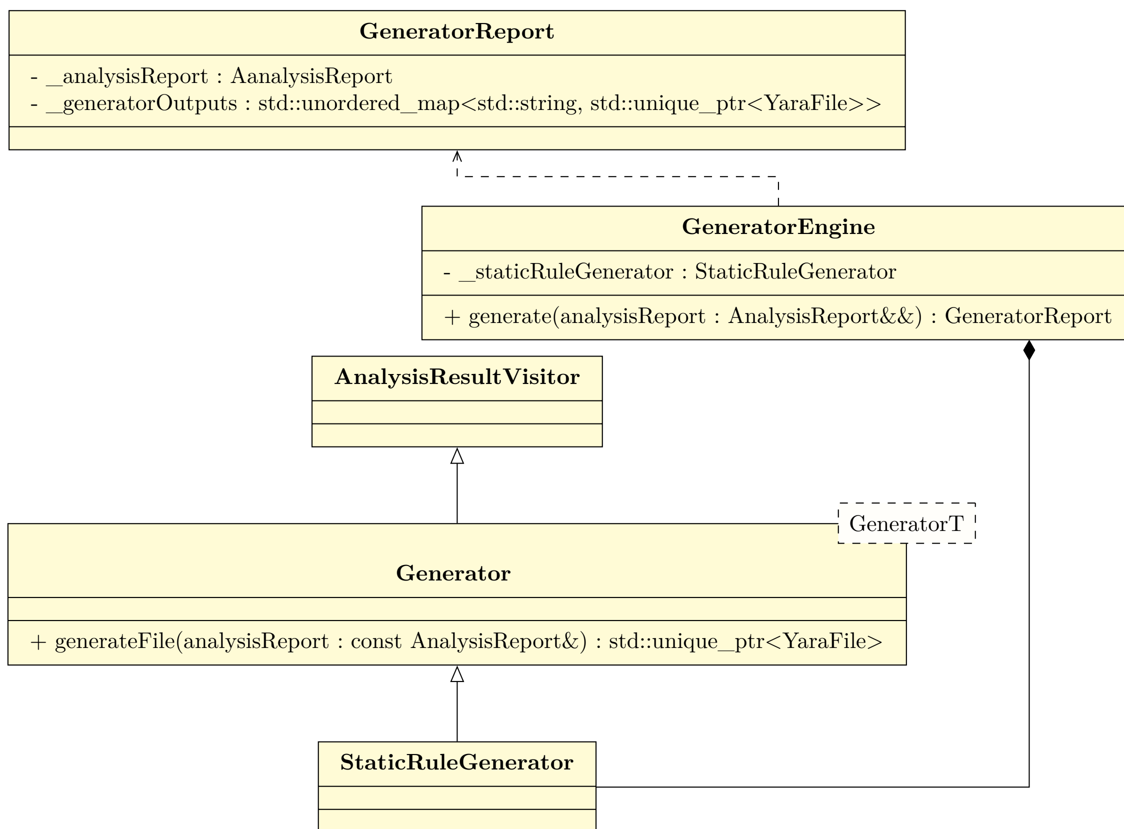
6.2.3 Generátor

Generátor je implementovaný v triede `GeneratorEngine`, ktorej metóda `generate` tvorí základ generovania. Táto metóda očakáva `AnalysisReport`, teda výsledky všetkých analýz. Jednotlivé moduly generátoru sú tvorené abstraktnou triedou `Generator` dediacou z `AnalysisResultVisitor`. Zatiaľ existuje len jediný modul a to je `StaticRuleGenerator`, generujúci statické YARA pravidlo. `GeneratorEngine` obsahuje v sebe instanciu tohto modulu a stará sa o jeho spustenie pomocou metódy `generateFile`.

`StaticRuleGenerator` využíva knižnicu *yaral* na zostrojovanie podmienok jednotlivých analýz v implementovaných `visit` metódach. Na pridanie podmienky sa na konci každej

`visit` metódy zavolá `addConditionTerm`, ktorý sa postará o pridanie do správnej časti podmienky (restriktívnej resp. neristrikívnej).

Výsledkom generátoru je `GeneratorReport`. Ten obsahuje ako pôvodný `AnalysisReport`, z ktorého sa generovali informácie, tak aj tabuľku vygenerovaných `yaral::YaraFile` od odpovedajúcich generačných modulov.



Obr. 6.4: Zjednodušený diagram tried pre generátor.

6.2.4 Konfigurácia

Trieda `Config` predstavuje konfiguráciu celého nástroja. Táto konfigurácia sa inicializuje v triede `ConfigFileParser`, ktorá sa stará o spracovanie konfiguračného súboru v INI formáte. Načítavanie tohto súboru sa robí pomocou modulu `boost::property_tree`.

Pokiaľ niektorá z tried v celom systéme chce využívať konfiguráciu, tak musí dediť z triedy `UsesConfig`, čím sa zaväzuje, že poskytne konštruktor, ktorý prijíma referenciu na konštantnú instanciu triedy `Config`. V rámci danej triedy je následne dostupná metóda `getConfig` na prístup k tomuto objektu.

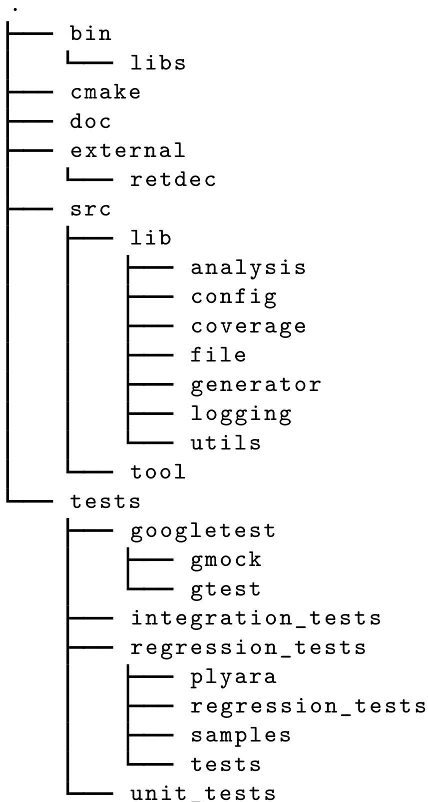
6.3 Zdrojové kódy, metriky, kompilácia a spúšťanie

Táto podkapitola popisuje spôsob uloženia zdrojových kódov, jeho metriky a ponúka návod ako preložiť a pracovať s nástrojom *yaragen*.

6.3.1 Zdrojové kódy

Vo verzií nástroja, ktorá sa používa interne v AVG Technologies a je verzovaná systémom Git je adresárová štruktúra mierne odlišná od adresárovej štruktúry v rámci tejto práce. Prístup k zdrojovým kódom rekonfigurovateľného spätného prekladača nie je dostupný verejnosti a preto je nutné tieto zdrojové kódy sprístupniť priamo. Základná adresárová štruktúra celého projektu je zobrazená na ukážke 6.2.

Kód 6.2: Adresárová štruktúra projektu.



```
.
├── bin
│   └── libs
├── cmake
├── doc
├── external
│   └── retdec
├── src
│   ├── lib
│   │   ├── analysis
│   │   ├── config
│   │   ├── coverage
│   │   ├── file
│   │   ├── generator
│   │   ├── logging
│   │   └── utils
│   └── tool
└── tests
    ├── googletest
    │   ├── gmock
    │   └── gtest
    ├── integration_tests
    ├── regression_tests
    │   ├── plyara
    │   ├── regression_tests
    │   ├── samples
    │   └── tests
    └── unit_tests
```

Adresár `bin` slúži ako priestor štandardnú pre inštaláciu preloženého nástroja a knižníc. Súbory súvisiace s nástrojom CMake [4] sú umiestnené v zložke `cmake`. Zložka `doc` obsahuje súbory potrebné pre generovanie dokumentácie pomocou nástroja *doxygen*. Rekonfigurovateľný spätný prekladač je v zložke `external`, ako externá závislosť. Adresár `src` obsahuje dve ďalšie adresáre `lib` a `tool`. V tom prvom sa nachádza implementácia knižnice `yaragenlib`. Je taktiež členený na ďalšie podadresáre pre prehľadnosť zdrojových kódov. V adresári `tool` je zase implementácia samotného nástroja `yaragen`. Testy a všetko s nimi súvisiace sa nachádza v adresári `tests`. Testy sú popísané ďalej v práci v kapitole 7.

6.3.2 Metriky

Medzi metriky, ktoré skúmame o danom zdrojovom kóde sú počet súborov, riadkov kódu, komentárov a prázdne riadky. Tieto metriky boli vygenerované nástrojom `cloc` [1]. Metriky pre zložku `src` sú:

- Počet súborov: 81
- Riadky s komentármi: 2086

- Riadky s kódom: 4563
- Prázdne riadky: 1139
- Riadkov celkovo: 7788

Metriky pre jednotkové a integračné testy sú:

- Počet súborov: 11
- Riadky s komentármi: 48
- Riadky s kódom: 1664
- Prázdne riadky: 302
- Riadkov celkovo: 2014

6.3.3 Kompilácia

Nástroj *yaragen* je určený ako pre systémy Windows, tak aj Linux. Konkrétne podporované prekladače sú GCC a Clang s podporou C++14 na platforme Linux a Microsoft Visual C++ od verzie 2015 pre platformu Windows. To by vyžadovalo spravovať súbory pre kompiláciu na oboch platformách, čo je náchylné na chyby a vyžaduje dvojitú modifikáciu v prípade zmien v kompilácií.

Z toho dôvodu bol zvolený nástroj CMake [4] umožňujúci generovanie potrebných súborov pre kompiláciu pre rôzne prekladače na rôznych platformách. Základným stavebným prvkom sú CMake súbory, ktoré sa umiestnia do adresárovej štruktúry projektu a následne je možné špeciálnym deklaratívnym jazykom popísať, čo všetko sa má prekladať, akým spôsobom a čo má byť výsledok prekladu.

6.3.4 Spustenie

Program `yaragen` pri spustení očakáva zoznam súborov alebo adresárov, ktoré rekurzívne prehľadáva. Tieto súbory sú pre determinizmus pri vymenenom poradí vstupných súborov zoradené abecedne. V prípade spúšťania programov je však často zavedený limit pre počet znakov parametrov, či priamo počet parameterov. Program `yaragen` je však určený na spúšťanie nad veľkými množinami súborov a preto je tento limit obmedzujúci. Pri spustení bez parametrov sa preto očakávajú novým riadkom oddelené cesty k adresárom alebo súborom, až po `EOF`.

Pri spustení je taktiež možné poskytnúť dodatočné prepínače, ktoré upravia chod programu. Na ich spracovanie bol použitý modul `boost::program_options`. Celkovú implementáciu načítavania prepínačov pri spustení je možné nájsť v triede `CommandLineParser`. Všetky dostupné možnosti sú znázornené v tabuľke 6.1.

Názov	Parametre	Popis
`-h/--help`	-	Zobrazí nápovedu.
`-q/--quiet`	-	Vypne zobrazovanie stavových správ.
`-l/--log-level`	Úroveň	Nastaví úroveň stavových správ. Dostupné úrovne: `debug`, `info`, `warning`, `error`, `fatal`. Prednastavené: `warning`.
`-c/--config`	Cesta	Cesta ku konfiguračnému súboru.
`-a/--analyses`	ID1,ID2,...	Identifikátory analýz, ktoré vykonať.
`-e/--exclude`	ID1,ID2,...	Identifikátory analýz, ktoré vynechať.
`--rule-name`	Názov	Názov pravidla.
`--author`	Autor	Autor pravidla.
`--desc`	Popis	Popis pravidla.
`--rel`	Spoľahlivosť	Spoľahlivosť pravidla.
`--strain`	Rodina	Rodina pravidla.
`--type`	Typ	Typ pravidla.
`--meta`	Kľúč=hodnota,...	Vlastné meta informácie pravidla.
`-o/--output`	Cesta	Cesta k výstupnému súboru. Prednastavený je štandardný výstup.

Tabuľka 6.1: Dostupné parametre programu `yaragen`.

6.4 Optimalizácie

Počas integrácie knižnice *fileformatl* sa zistilo, že generovanie detekčného vzoru trvá neprimerane dlho (viac ako 30 sekúnd na súbor). Po profilovaní bolo zistené, že najviac času sa trávi práve v knižnici *fileformatl*, pri počítaní hashov SHA256 a SHA1. Tie sa počítajú zo sekcií, importovaných symbolov, *resourcov* (ikony, obrázky a iné). To je spôsoboné tým, že táto knižnica načíta zo vstupného súboru všetky dostupné informácie, bez ohľadu na to či sú potrebné alebo nie.

Z toho dôvodu boli zavedené do knižnice *fileformatl* tzv. príznaky načítavania (ďalej označované ako *loader flags*), ktoré umožňujú vypnúť či zapnúť načítavanie niektorých vlastností, pokiaľ nie sú potrebné. Implementované *loader flags* je možné vidieť v tabuľke 6.2.

Názov	Hodnota	Popis
`NONE`	`0x0`	Načítava sa všetko ako obyčajne.
`NO_FILE_HASHES`	`0x1`	Nepočíta sa hash súboru.
`NO_VERBOSE_HASHES`	`0x2`	Nepočítajú sa zvyšné hashe (sekcie, importy, ...)
`DETECT_STRINGS`	`0x4`	Detekujú sa reťazce, ktoré sa štandardne nedetekujú.

Tabuľka 6.2: *Loader flags* dostupné v knižnici *fileformatl*.

Kapitola 7

Testovanie a výsledky

Vytvorený nástroj je testovaný na niekoľkých úrovniach. Táto kapitola popisuje spôsoby testovania a vyhodnocuje kvalitu vytvorených detekčných vzorov. Na záver tejto kapitoly je ukážka použitia nástroju v praxi.

7.1 Testovanie generátoru

Testovanie generátoru je robené formou jednotkových (*unit*) testov, integračných testov a regresných testov. V nasledujúcich podkapitolách sú popísané jednotlivé druhy testov, na čo sa sústreďujú a čo je ich cieľom.

7.1.1 Jednotkové testy

Úlohou jednotkových testov je otestovať samostatné funkčné jednotky nezávisle od okolného prostredia. Programovací jazyk pre jednotkové testy bol zvolený taktiež C++. Použitá knižnica *googletest* [8] tento proces zjednodušuje použitím niekoľko makier preprocesora.

U jednotkových testov je kladený dôraz hlavne na triedy, ktorých vstup nezávisí na platforme (súborový systém, vstup/výstup atď.), alebo tiež nezávisí na výstupoch iných častí generátoru. Medzi testované jednotky resp. triedy patria nasledovné:

- `CoverageResolver`
- `EntryPoint`
- `ImportSymbol`
- `RichHeader`
- `Section`
- `String`

Cieľom týchto testov je odhaliť zmenu v kľúčovom správaní danej jednotky, čo by mohlo mať vplyv na nesprávne správanie celého systému. Pri overovaní testov je vhodné ich spúšťať ako prvé, aby sa chyby nepreniesli do testov vyššej úrovne.

7.1.2 Integračné testy

Integračné testy majú za úlohu testovať jednotlivé jednotky prepojené resp. integrované dohromady. Tak isto ako u jednotkových testov je použitá knižnica *googletest* spolu s rozšírením *googlemock* [8], ktoré umožňuje vytvárať *mock* objekty, ktoré dokážu nahradiť správanie nejakej triedy úplne iným správaním s absolútne rovnakým rozhraním. Tým eliminujeme potrebu prítomnosti skutočných súborov pri testovaní.

Zameriavame sa hlavne na integráciu funkčnosť spolupráce analyzátoru a generátoru. V rámci vývoja týchto testov boli vytvorené vlastné makrá, ktoré umožňujú písať integračné testy veľmi rýchlo a efektívne. V prípade, že by chceme vytvoriť nový integračný test, tak by sme postupovali podľa nasledovných krokov.

Extrakcia

Najskôr je nutné poskytnúť akciu, ktorú vykoná *mock* objekt extraktoru. Na to nám slúžia tri dostupné makrá.

- `LOADER_FAIL` – definuje, že extraktor zlyhá a nebudú extrahované žiadne informácie.
- `LOADER_FAIL_INVOKE(fn)` – definuje, že extraktor zlyhá, ale navyše sa vykoná lambda funkcia `fn`. Volaná funkcia musí prijímať referenciu na `FileInformation`. Toto makro je použiteľné, ak napríklad chceme zadať falošnú cestu alebo formát súboru, pre ktorý extrakcia zlyhala.
- `LOADER_SUCCESS(times, fn)` – definuje, že extraktor uspeje, pričom vykoná lambda funkciu `fn` presne `times` krát. Pre túto funkciu platí to isté ako u `LOADER_FAIL_INVOKE`. Viacnásobné spustenie funkcie je použité na simulovanie extrakcia niekoľkých súborov.

Konfigurácia

Pomocou premennej `_config` je možné následne poskytnúť vlastnú konfiguráciu. Tento krok nie je povinný, takže je možné ponechať aj štandardné nastavenia konfigurácie. Použiteľné je to v prípade, že chceme nastaviť hodnoty pre meta informácie výstupného pravidla, alebo prekonfigurovať analýzy.

Na rekonfiguráciu analýz je lepšie použiť metódu `prepareAnalysisSettings`, ktorej je možné priamo poskytnúť konfiguračné možnosti a ich hodnoty, ako je to možné spraviť v konfiguračnom súbore popísanom v kapitole 5.2.4.

Kontrola YARA pravidla

Posledným krokom je kontrola vygenerovaného YARA pravidla, či odpovedá očakávaniam. Táto kontrola začína makrom `START_YARA_FILE_CHECK`. Nasleduje zoznam `EXPECT` makier pre rôzne časti YARA súboru.

- `EXPECT_HEADER(ruleCount, coveredFiles)` – Podmienka pre úvodné meta informácie v komentári, ktoré boli popísané v kapitole 5.2.3. `ruleCount` značí očakávaný počet verejných pravidiel a `coveredFiles` celkový počet pokrytých súborov.
- `EXPECT_MSIL_RULE` – Značí, že sa očakáva súkromné MSIL pravidlo v prípade, že sú všetky súbory vo formáte .NET.
- `EXPECT_IMPORTS(...)` – Očakávaný zoznam importovaných YARA modulov.

- `EXPECT_AUTHOR(author)` – Očakávaný autor v meta informáciach YARA pravidla.
- `EXPECT_DESCRIPTION(desc)` – Očakávaný popis v meta informáciach YARA pravidla.
- `EXPECT_RELIABILITY(rel)` – Očakávaná spoľahlivosť v meta informáciach YARA pravidla.
- `EXPECT_STRAIN(strain)` – Očakávaná rodina *malwaru* v meta informáciach YARA pravidla.
- `EXPECT_TYPE(type)` – Očakávaný typ *malwaru* v meta informáciach YARA pravidla.
- `EXPECT_META(k, v)` – Ľubovoľná očakávaná meta informácia v YARA pravidle s názvom `k` a hodnotou `v`.
- `EXPECT_STRINGS(...)` – Zoznam očakávaných reťazcov v YARA pravidle.
- `EXPECT_CONDITION(cond)` – Očakávaná podmienka YARA pravidla. Musí sa jednať o správne odsadenú podmienku.

Na poradí jednotlivých makier nezáleží, ale je vhodné postupovať v takom poradí, v akom sa nachádzajú v súbore. Tento blok je zakončený makrom `END_YARA_FILE_CHECK`. Príklad takéhoto testu je zobrazený na ukážke 7.1.

Kód 7.1: Ukážka integračného testu.

```
TEST_F(IntegrationTests,
PeFileWithEntryPoint) {
    LOADER_SUCCESS(1, [](FileInformation& fi) {
            fi.setFormatFlag(FormatFlag::Pe);
            fi.setEntryPoint({ 0x400000, std::vector<std::uint8_t>{ 0x01,
               ↪ 0xFF, 0x02, 0xFE } });
        });

    prepareAnalysisSettings<EntryPointAnalysis>(
            "min_percent_coverage", "0",
            "must_cover_all", "1",
            "restrictive", "0",
            "bytes_count", "4");

    START_YARA_FILE_CHECK;
    EXPECT_HEADER(1, 1);
    EXPECT_IMPORTS("pe");
    EXPECT_STRINGS("$entry_point_g0_p100 = { 01 FF 02 FE }");
    EXPECT_CONDITION(R"y(
        ((pe.entry_point == 0x400000 and $entry_point_g0_p100 at
           ↪ pe.entry_point)))y");
    END_YARA_FILE_CHECK;
}
```

7.1.3 Regresné testy

Pri integračných testoch nie je šanca zachytiť všetky možnosti vstupných hodnôt, ktoré môžu reálne súbory v sebe obsahovať a nebolo by ich jednoduché nasimulovať pomocou dostupných nástrojov. Občas je nutné aby boli testy vykonané nad reálnymi súbormi. Na to slúžia regresné testy. Ich úlohou je zachytiť špecifický prípad toho, že pre danú množinu

súborov funguje *yaragen* očakávane a v prípade ďalších zmien nedôjde k regresií a tento špeciálny prípad prestane fungovať. Narozdiel od jednotkových a integračných testov, ktoré sa tvoria hlavne v prípade, keď sú pridávané nové funkčnosti, tak regresné testy sa vytvárajú na základe ohlasovaných chýb nad konkrétnymi množinami súborov.

Regresné testy už nie sú písané v jazyku C++, ale dochádza v nich k ozajstnému spusteniu nástroja *yaragen*. Aby šlo tento proces nejak automatizovať, tak bol vyvinutý jednoduchý testovací *framework* v jazyku Python vo verzií 3.6 [17]. Tento *framework* používa modul `unittest` a modul tretej strany, `plyara`, podporujúci parsovanie pravidiel vo formáte YARA. Tento *framework* ponúka možnosť ako jednoducho spustiť *yaragen* a získať výsledok pomocou metódy `run_yaragen` vracajúcu dvojicu YARA súbor a chybové hlásenia. Očakáva sa však, že *yaragen* je dostupný v premennej prostredia, teda je spustiteľný cez názov *yaragen* od všadiaľ. Príklad ukážkového regresného testu je možné vidieť na ukážke 7.2.

Kód 7.2: Ukážka regresného testu.

```
import regression_tests

class Test(regression_tests.Test):
    def test_imports_by_names_are_prioritized_over_imports_by_ordinal(self):
        yara_file, err = self.run_yaragen(['samples/
            ↪ test_import_name_prioritization'])

        self.assertEqual(1, len(yara_file.rules))
        self.assertTrue(yara_file.rules[0].condition_contains(r'pe\.imports
            ↪ \("KERNEL32.dll",␣"[^"]+"\)'))
        self.assertFalse(yara_file.rules[0].condition_contains(r'pe\.imports
            ↪ \("KERNEL32.dll",␣[0-9]+\)'))
```

Spustiť regresné testy je možné hocijakým nástrojom na spúšťanie Python testov, ako sú napríklad `nose2` alebo `green`.

7.2 Vyhodnotenie kvality detekčných vzorov

Vyhodnocovanie prebehlo na súboroch v zhlukoch, ktoré vytvoril Clusty. Zobralo sa maximálne 100 náhodných, dostatočne veľkých (u PE minimálne 100 súborov, u .NET 20) zhlukov. Pre každý zhluk bol vytvorený detekčný vzor pomocou nástroja *yaragen*. Následne sa zobrali všetky vygenerované detekčné vzory a vyhľadali sa v každom súbore každého zhluku a zaznamenalo sa, ktoré detekčné vzory sa v tom súbore našli. Na to bol použitý nástroj *yara*. Pre zrýchlenie prehľadávania bol každý detekčný vzor preložený do bajtkódu pomocou nástroja *yarac*. Pri vyhodnocovaní skúmame hlavne chyby typu 1 (*false positive*), teda koľko krát došlo k detekcií, keď k nej nemalo dôjsť.

Výsledky sa nachádzajú v tabuľke 7.1. Testovacia sada neobsahovala ani jeden prípad, kedy by došlo k nesprávnemu zaradeniu súboru do zhluku. K tomuto výsledku dopomohol systém Clusty, ktorý vytvoril kvalitné zhluky súbory, ale tiež precízne zvolené vlastnosti používané pri tvorbe detekčných vzorov.

Formát	Počet zhlukov	Počet súborov	Počet chýb
PE	100	13768	0
.NET	68	2845	0

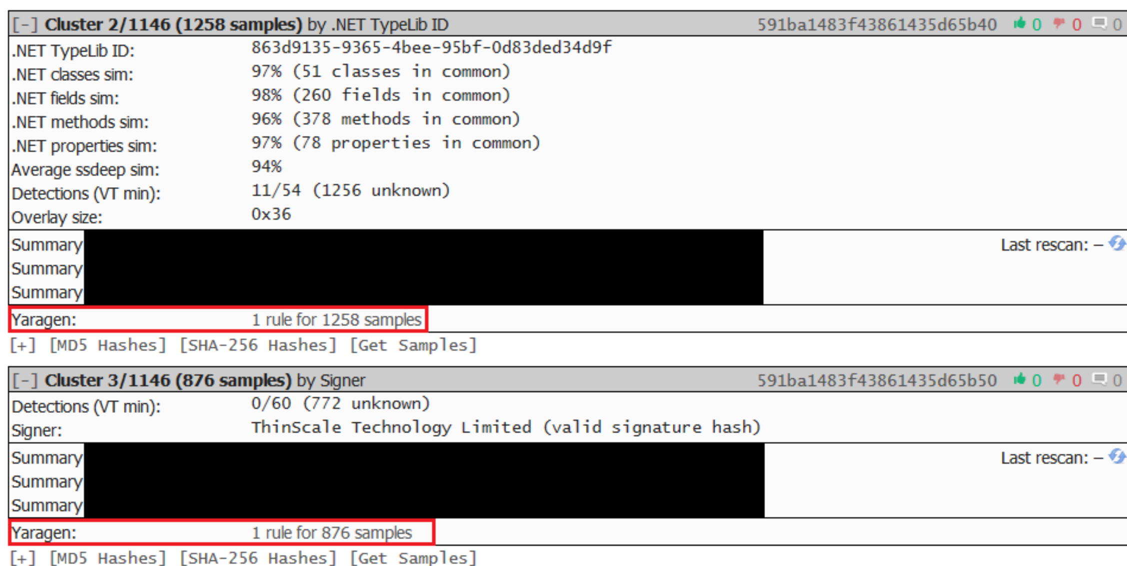
Tabuľka 7.1: Výsledky vyhodnotenia kvality detekčných vzorov.

7.3 Integrácia a nasadenie v praxi

Táto kapitola popisuje príklady použitia nástroja *yaragen* v praxi a niektoré štatistiky, ktoré s tým súvisia.

7.3.1 Clusty

Integrácia do systému Clusty je vo forme spustenia *yaragen* pre každý zhluk. Vygenerované detekčné vzory sú uložené do databáze. Na obrázku 5.1 je možné vidieť akým spôsobom je tento detekčný vzor sprístupnený formou webového rozhrania. V poslednom riadku každého zhluku sa nachádza položka `Yaragen: X rule(s) for XYZ samples`. Jedná sa len o zobrazenie meta informácií z úvodného komentára výstupného súboru. Po kliknutí na tento text sa ponúkne užívateľovi dialóg na stiahnutie vygenerovaného YARA pravidla.



Obr. 7.1: Nástroj *yaragen* integrovaný do webového rozhrania systému Clusty.

Štatistiky behu *yaragenu* v Clustym obsahujú počet zhlukov nad ktorými bol spúšťaný, dobu behu, počet nedokončených behov z dôvodu prekročenia 15 minút a počet zhlukov pre ktoré sa nepodarilo vygenerovať žiadne pravidlo. Spúšťanie prebiehalo v 72 paralelných procesoch na dvoch procesoroch Intel Xeon E5-2699v3@2.3 GHz so 64-bitovým systémom Debian Jessie. Clusty je spúšťaný denne, takže každý riadok tabuľky 7.2 znamená jeden deň.

Počet zhlukov (PE + .NET)	Počet súborov (PE + .NET)	Doba behu	Žiadne pravidlo	Nedokončený beh
5504 + 1312	231887 + 24027	2h 00m 45s	885	145
5562 + 1146	291018 + 21755	2h 11m 32s	647	173
5480 + 1295	312251 + 21670	2h 01m 10s	691	148
4896 + 944	297205 + 21814	1h 53m 20s	491	132
5742 + 1211	316814 + 23742	2h 09m 02s	758	152
5544 + 1053	318344 + 21877	1h 57m 39s	623	153
5952 + 1789	342327 + 27289	2h 08m 13s	854	167

Tabuľka 7.2: Štatistiky behu *yaragenu* v systéme Clusty.

Priemerný počet zhlukov bol 6776 (5526 pre PE, 1250 pre .NET) a priemerný počet súborov 324574 (301407 pre PE, 23168 pre .NET). Priemerná doba behu bola 2h 03m 06s. Priemerne sa nevygenerovalo žiadne pravidlo pre 707 zhlukov a *yaragen* neskončil s úspechom 152 krát. To predstavuje, že približne pre 12% zhlukov sa nevygenerovalo žiadne pravidlo. To je spôsobené neprítomnosťou vhodnej statickej vlastnosti,

7.3.2 Ransomware WannaCry

Dňa 12. mája 2017 sa po svete začala šíriť nákaza menom *WannaCry*. Jedná sa o *ransomware* zneužívajúci chybu v systémoch Windows, konkrétne v implementácií SMB protokolu, ktorá povoluje vzdialené vykonávanie kódu. Nakazených bolo vyše 200 tisíc systémov a útočníkom bolo vyplatených ku dňu 20.5.2017 už vyše 100 tisíc dolárov vo forme *bitcoinov* [30, 29, 9].

V nasledujúcich riadkoch si ukážeme, ako je možné použiť *yaragen* na vytvorenie detekčného vzoru z reálneho *malware*. Využijeme zhluk, ktorý nám vytvoril Clusty z dňa 12. mája 2017 kedy začali prichádzať prvé vzorky, obsahujúci 128 súborov využiteľné na vygenerovanie detekčného vzoru. Skrátené vygenerované pravidlo je zobrazené na ukážke 7.3. Obsahuje totiž moc nič nehovoriacich reťazcov, preto sú uvedené len tie významné, ako sú napríklad *bitcoin* peňaženky.

Vytvorené pravidlo bolo použité v službe Retrohunt, ktorú ponúka VirusTotal ako časť svojho prémiového balíčka Intelligence. Retrohunt dovoľuje aplikovať YARA pravidlo na vzorky, ktoré boli zaslané do služby VirusTotal za posledné približne tri mesiace. V prípade pravidla pre WannaCry sa jednalo o súbory s veľkosťou 76,3 TB. Presný počet nie je dostupný cez webové rozhranie. Výsledok analýzy obsahoval 718 vzoriek, ktoré sa podarilo úspešne detekovať našim YARA pravidlom. Pri vyhodnocovaní výsledkov, súbory označené antivírom Avast ako WannaCry sa ignorovali a považovali automaticky za WannaCry. Pre tie zvyšné sa manuálne vyhodnotilo či sa o WannaCry jedná alebo nie. V 697 prípadoch sa jednalo o *ransomware* WannaCry, u ostatných 21 prípadov nastala chyba typu 1 (*false positive*). Úspešnosť je 97%.

Kód 7.3: Skrátené pravidlo pre *ransomware* WannaCry.

```
// Rules: 1
// Covered files: 128

import "pe"

rule rule_wannacry_static {
meta:
  author = "yaragen␣0.2"
  description = "Rule␣for␣WannaCrypt␣ransomware"
  reliability = "susp"
  strain = "WannaCry"
  type = "ransomware"
strings:
  $entry_point_g0_p100 = { 55 8B EC 6A FF 68 A0 A1 40 00 68 A2 9B 40 00 64
      ↪ A1 00 00 00 00 50 64 89 25 00 00 00 00 83 EC 68 53 56 57 89 65 E8
      ↪ 33 DB 89 5D FC 6A 02 FF 15 C0 A0 40 }
  // more strings
  $string_g1_025_p099 = "115p7UMMngoj1pMvkpHijcRdfJNXj6LrLn"
  $string_g1_026_p099 = "12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw"
  $string_g1_027_p099 = "13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94"
  // more strings
condition:
  ((pe.imports("MSVCP60.dll", "??0_Lockit@std@@QAE@XZ") and pe.imports("
      ↪ MSVCP60.dll", "??1_Lockit@std@@QAE@XZ") and /* more imports */ or
  (pe.rich_signature.clear_data == "DanS\x00\x00\x00\x00\x00\x00\x00\x00\x00
      ↪ \x00\x00\x00{\x1c\x0c\x00\x01\x00\x00\x00\x83\x1c\x0e\x00\x04\x00\
      ↪ x00\x00o\x1f\n\x00\x0b\x00\x00\x00o\x1f\x0b\x00\x01\x00\x00\x00o\
      ↪ x1f\x04\x00\x04\x00\x00\x00\xc3\x0f]\x00\x0b\x00\x00\x00\x00\x00\
      ↪ x01\x00[\x00\x00\x006&\x0b\x00\x01\x00\x00\x00\xc7\x06\x06\x00\x01\
      ↪ x00\x00\x00") or
  ((pe.entry_point == 0x9a16 and $entry_point_g0_p100 at pe.entry_point)) or
  (all of ($string_g0*) or all of ($string_g1*) or all of ($string_g2*) or (
      ↪ $string_g3_0_p093 and $string_g3_1_p093 and $string_g3_2_p093 and
      ↪ $string_g3_3_p093) or all of ($string_g4*) or all of ($string_g5*))
}
```

Kapitola 8

Záver

V tejto práci sme sa zaoberali vytvorením systému na tvorbu detekčných vzorov z binárnych súborov. Práca bola vypracovaná v spolupráci so spoločnosťou AVG Technologies. Tvorba takého systému vyžadovala návrh niekoľkých nových analýz na extrakciu informácií zo spustiteľných súborov do existujúcej knižnice *fileformatl*, ktorá bola použitá na spracovania spustiteľných súborov. Jednalo sa konkrétne o analýzy na extrakciu certifikátov, digitálneho podpisu, rekonštrukciu dátových typov u .NET súborov a detekciu reťazcov. Nové, ale aj existujúce analýzy boli použité na návrh a implementáciu nástroja, ktorý generuje detekčný vzor a nazýva sa *yaragen*.

Vytvorený nástroj slúži na popis zhlukov súborov pomocou detekčného vzoru zapísaného v deklaratívnom jazyku YARA. Detekčný vzor slúži na rýchle priraďovanie nových spustiteľným súborov k už existujúcim rodinám *malwaru*, a to bez nutnosti vykonávania zdĺhavej zhlukovej analýzy. Tým dosahujeme lepšiu efektivitu pri analýze *malwaru*. Jednotný formát zápisu detekčného vzoru je taktiež podstatný fakt, pretože dovoľuje jednoduchú spoluprácu so službou VirusTotal, ktorá je frekventovane využívaná, tak ako to bolo zobrazené v tejto práci. To prináša so sebou lepšiu responzívnosť v prípade nových hrozieb a dovoľuje okamžitú tvorbu prvotného detekčného vzoru a získavanie ďalších vzoriek z rovnakej rodiny.

Na implementáciu bol použitý jazyk C++ v štandarde ISO C++14. Spolu s nástrojom boli vytvorené jednotkové, integračné a regresné testy. Novovzniknutý nástroj sa používa v praxi a je integrovaný do systému pre zhlukovú analýzu Clusty, kde sa denne spúšťa nad viac ako 300 tisíc súbormi. Taktiež je využívaný analytikmi v spoločnosti AVG Technologies pri hľadaní nových vzoriek. Ako ukázali výsledky, tak *yaragen* v prípade spolupráce so systémom Clusty dokáže vytvoriť dostatočne jedinečné detekčné vzory s veľmi minimálnou až nulovou chybovosťou. Praktickou ukážkou na *ransomwari* WannaCry tiež bolo ukázané, že si vie poradiť aj s aktuálnym *malwarom* a dokáže s vysokou úspešnosťou zdetekovať nové vzorky. Dosiahnutá úspešnosť bola 97%.

Ciele, ktoré sme si v úvode tejto práce stanovili, boli splnené. Pribudli nové druhy analýz pre extrakciu informácií zo súborov, ktoré začal využívať aj systém Clusty, čim sa zbavil závislosti na riešeniach tretích strán, ako je napríklad .NET *disassembler* pre rekonštrukciu dátových typov pre .NET platformu. Druhým splneným cieľom je hotový nástroj, ktorý automaticky dokáže vygenerovať detekčný vzor.

Do budúcnosti je možné nástroj *yaragen* rozšíriť o mnoho ďalších vylepšení. Pri návrhu bol braný ohľad na modulárnosť riešenia, takže pridávanie nových analýz je veľmi jednoduché. V rámci tejto práce bol generovaný detekčný vzor zložený len zo statických vlastností, avšak je viac ako nutné pridať podporu aj pre dynamické vlastnosti. Ďalšou možnosťou je kaskádovanie detekčných vzorov podľa toho, ktoré analýzy produkujú lepšie výsledky. To

súvisí s automatizovanou konfiguráciou jednotlivých analýz, ktorá by potenciálne mohla prejsť až k riešeniu so strojovým učením. Táto práca sa primárne venovala platforme Windows, ale nesmieme však zabudnúť aj na iné platformy ako Linux, Mac OS X, Android, či iné formáty spustiteľných súborov ako ELF, Mach-O, DEX a mnoho ďalších. Na túto prácu bude nadväzovať dizertačná práca, ktorej cieľom bude vylepšiť riešenia v tejto práci a priniesť inovatívne metódy, ako vytvárať detekčné vzory.

Literatúra

[1] AlDanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages. [cit. 2017-05-21].
URL https://github.com/AlDanial/cloc

[2] Automated Malware Analysis - Cuckoo Sandbox. [cit. 2017-01-05].
URL https://cuckoosandbox.org/

[3] Boost C++ Libraries. [cit. 2017-05-13].
URL http://www.boost.org/

[4] CMake. [cit. 2017-05-21].
URL https://cmake.org/

[5] dotnet module — yara 3.6.0 documentation. [cit. 2017-05-20].
URL http://yara.readthedocs.io/en/v3.6.0/modules/dotnet.html

[6] FNV Hash. [cit. 2017-05-13].
URL http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1a

[7] Fuzzy Hashing and ssdeep. [cit. 2017-01-05].
URL http://ssdeep.sourceforge.net/

[8] google/googlemock: Google Mock. [cit. 2017-05-20].
URL https://github.com/google/googlemock

[9] How Much Wannacry Paid the hacker. [cit. 2017-05-20].
URL http://howmuchwannacrypaidthehacker.com/

[10] ISO/IEC 14882:2014 - Information technology – Programming languages – C++. [cit. 2017-05-20].
URL https://www.iso.org/standard/64029.html

[11] Malware is being signed with multiple digital certificates to evade detection. [cit. 2017-01-05].
URL https://www.symantec.com/connect/blogs/malware-being-signed-multiple-digital-certificates-evade-detection

[12] OpenSSL: Cryptography and SSL/TLS Toolkit. [cit. 2017-01-07].
URL https://www.openssl.org/

[13] PeLib - An open-source C++ library. [cit. 2017-01-07].
URL http://www.pelib.com/index.php

[14] Retargetable Decompiler. [cit. 2017-01-05].
URL https://retdec.com/

[15] Signed Malware: You Can Run, But You Can't Hide. [cit. 2017-01-05].
URL https://securingtomorrow.mcafee.com/mcafee-labs/signed-malware-you-can-runbut-you-cant-hide/

[16] VirusTotal - Free Online Virus, Malware and URL Scanner. [cit. 2017-01-05].
URL https://virustotal.com/

[17] What's New In Python 3.6 - Python 3.6.1 documentation. [cit. 2017-05-20].
URL https://docs.python.org/3/whatsnew/3.6.html

[18] YARA 3.5.0 documentation. [cit. 2017-01-05].
URL https://yara.readthedocs.io/en/v3.5.0/

[19] AV-TEST: SECURITY REPORT 2015/2016. [cit. 2017-01-04].
URL https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf

[20] Blaszczyk, A.: PE Section names – re-visited. [cit. 2017-05-15].
URL
http://www.hexacorn.com/blog/2016/12/15/pe-section-names-re-visited/

[21] Cooper, D.; NIST; Santesson, S.; aj.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280.
URL https://tools.ietf.org/html/rfc5280

[22] ECMA International: *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, June 2012.
URL
http://www.ecma-international.org/publications/standards/Ecma-335.htm

[23] Eilam, E.; Chikofsky, E. J.: *Reversing: Secrets of Reverse Engineering*. Indianapolis: Wiley, 2005, ISBN 0-7645-7481-7.

[24] Emm, D.; Unuchek, R.; Garnaeva, M.; aj.: IT THREAT EVOLUTION IN Q3 2016. [cit. 2017-01-04].
URL https://securelist.com/files/2016/11/KL_Q3_Malware_Report_ENG.pdf

[25] International Telecommunication Union: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T X.680.
URL http://www.itu.int/itu-t/recommendations/rec.aspx?rec=x.680

[26] International Telecommunication Union: Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). ITU-T X.690.
URL http://www.itu.int/itu-t/recommendations/rec.aspx?rec=x.690

[27] International Telecommunication Union: The World in 2016: ICT Facts and Figures. [cit. 2016-12-28].
URL http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf

[28] Kaliski, B.: PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315.
URL https://tools.ietf.org/html/rfc2315

[29] Křoustek, J.: WannaCry ransomware that infected Telefonica and NHS hospitals is spreading aggressively, with over 50,000 attacks so far today. [cit. 2017-05-20].
URL https://blog.avast.com/ransomware-that-infected-telefonica-and-nhs-hospitals-is-spreading-aggressively-with-over-50000-attacks-so-far-today

[30] Křoustek, J.: WannaCry update: The worst ransomware outbreak in history. [cit. 2017-05-20].
URL https://blog.avast.com/wannacry-update-the-worst-ransomware-outbreak-in-history

[31] Microsoft Corporation: Microsoft Portable Executable and Common Object File Format Specification. [cit. 2017-01-04].
URL https://www.microsoft.com/en-us/download/details.aspx?id=19509

[32] Microsoft Corporation: Windows Authenticode Portable Executable Signature Format. [cit. 2017-01-04].
URL http://msdn.microsoft.com/en-us/windows/hardware/gg463180.aspx

[33] Milkovič, M.: *Univerzální nástroj pro dekompresi spustitelních souborů*. Bakalářská práce, Fakulta informačních technologií VUT v Brně, 2015.

[34] Pistelli, D.: Microsoft's Rich Signature (undocumented). [cit. 2017-05-18].
URL http://www.ntcore.com/files/richsign.htm

[35] Plaskoň, P.: *Rozšíření systému pro shlukovou analýzu binárních souborů*. Bakalářská práce, Fakulta informačních technologií VUT v Brně, 2017.

[36] Sikorski, M.; Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA, USA: No Starch Press, první vydání, 2012, ISBN 1593272901.

[37] Singh, S.: *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*. New York, NY, USA: Doubleday, první vydání, 1999, ISBN 0385495315.

[38] Vandevoorde, D.; Josuttis, N. M.: *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002, ISBN 9780201734843.

[39] Wallace, B.: Using .NET GUIDs to help hunt for malware. 2015, [cit. 2017-05-13].
URL https://www.virusbulletin.com/uploads/pdf/magazine/2015/vb201506-NET-GUIDs.pdf

[40] Wartell, R.; Zhou, Y.; Hamlen, K. W.; aj.: Differentiating Code from Data in x86 Binaries. University of Texas at Dallas.
URL http://www.utd.edu/~hamlen/wartell-pkdd11.pdf

[41] Zavoral, M.: *Rozšíření nástroje pro analýzu spustitelných souborů*. Bakalářská práce, Fakulta informačních technologií VUT v Brně, 2014.