
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# VYLEPŠENÍ GENEROVÁNÍ VZORŮ PRO DETEKCI ŠKODLIVÉHO KÓDU

**IMPROVED PATTERN GENERATION FOR DETECTION OF MALICIOUS CODE**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **MARTIN ŠTĚPÁNEK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. ZBYNĚK KŘIVKA, Ph.D.**
**SUPERVISOR**



**BRNO 2019**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

# Zadání bakalářské práce


Student: **Štěpánek Martin**
Program: Informační technologie
Název: **Vylepšení generování vzorů pro detekci škodlivého kódu**
**Improved Pattern Generation for Detection of Malicious Code**
Kategorie: Překladače

Zadání:

1. Studujte problematiku analýzy binárních souborů. Zaměřte se na analýzu spustitelných souborů.
2. Seznamte se s jazykem YARA, který slouží k zápisu detekčních vzorů.
3. Prostudujte stávající systém YaraGen pro tvorbu detekčních vzorů ve formátu YARA, vyvíjený společností Avast.
4. Navrhněte metody pro generování nových rysů binárních souborů, jak statických tak i behaviorálních, do výsledného detekčního vzoru a vylepšete generování již stávajících rysů.
5. Po domluvě s vedoucím a konzultantem implementujte metody navržené v předchozím bodě do systému YaraGen.
6. Vytvořené řešení důkladně otestujte sadou testů, včetně reálných programů, nebo vzorků potenciálně škodlivých programů.
7. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- Milkovič, M. *Systém pro detekci vzorů v binárních souborech*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- Sikorski, M., Honig, A. *Practical Malware Analysis: a Hands-On Guide to Dissecting Malicious Software*. San Francisco: No Starch Press, 2012.

Pro udělení zápočtu za první semestr je požadováno:

- První čtyři body zadání a rozpracování pátého bodu.

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

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Konzultant: Milkovič Marek, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 16. října 2018

# Abstrakt

Tato práce se zabývá automatickým generováním vzorů, které jsou využitelné pro detekci škodlivého kódu. Cílem je vytvořit nástroj, který bude pomáhat analytikům při odhalování a detekování malwaru. Jsou zhodnoceny postupy detekce malwaru používané společností Avast Software. Je představen nástroj YaraGen, který byl v rámci této práce zdokonalen. Je popsáno několik typů analýz, které byly pro nástroj YaraGen implementovány. Hlavním přínosem této práce jsou především analýzy behaviorálních rysů škodlivého kódu.



# Abstract

This thesis deals with an automatic pattern generation, that can be used for detection of malicious code. The aim of this thesis is to create a tool to help the analysts to detect malware. Approaches of malware detection used in Avast Software are reviewed. A tool called YaraGen, which was improved in this work, is presented. New analyses implemented for YaraGen are introduced. The main contribution of this thesis are behavioral analyses of a malicious code.



# Klíčová slova

YaraGen, YARA, detekce vzorů, malware

# Keywords

YaraGen, YARA, pattern recognition, malware


# Citace

ŠTĚPÁNEK, Martin. *Vylepšení generování vzorů pro detekci škodlivého kódu.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

# Vylepšení generování vzorů pro detekci škodlivého kódu

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl odborný konzultant Ing. Marek Milkovič a Ing. Jakub Křoustek, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Štěpánek
15. května 2019


## Poděkování

Rád bych poděkoval Ing. Zbyňku Křivkovi, Ph.D. za vedení práce a za konzultace. Dále bych chtěl poděkovat svému konzultantovi Ing. Marku Milkovičovi za všechny poskytnuté rady a za jeho práci na YaraGenu, na kterou jsem mohl navázat.

# Obsah

# Kapitola 1

# Úvod

Rozšíření počítačů a internetu v dnešní době usnadňuje lidem životy, má však i svou stinnou stránku. Stále častěji jsou tyto technologie totiž terčem útočníků, kteří přicházejí se stále sofistikovanějšími metodami, jak napadnout zařízení uživatelů. V boji proti těmto útočníkům je proto také potřeba používat moderní technologie a neustále tyto technologie vylepšovat.

Při vývoji programů chránících před škodlivým softwarem (malware z anglického malicious software) je třeba analyzovat, jaké má škodlivý software vlastnosti a jakým způsobem pracuje. Manuální analýza je však velice zdlouhavá a nehodí se pro velké objemy dat. Cílem této práce je poskytnout prostředek pro zautomatizování analýzy škodlivého kódu. Tato automatizace si neklade za cíl plně nahradit manuální analýzu, ale měla by být významnou podporou, která ulehčí práci analytikům a pomůže jim dosáhnout lepších výsledků v kratším čase.

Hlavním cílem je zdokonalení nástroje YaraGen pro automatické generování detekčních vzorů ve formátu YARA. Tento nástroj byl vytvořen Ing. Markem Milkovičem v rámci jeho diplomové práce vypracované ve spolupráci se společností AVG Technologies. Roku 2016 se z AVG Technologies stala dceřiná společnost Avast Software, takže tato bakalářská práce byla již vytvořena pod záštitou Avast Software.

Z tohoto důvodu budou v dalších kapitolách často zmiňovány technologie používané touto společností. Popsané principy jsou však obecnější a výsledný program může být použit zcela samostatně. Formát YARA, který je výstupním formátem nástroje YaraGen, je používán mnoha společnostmi v oblasti ochrany proti škodlivému softwaru (např. ESET, Kaspersky Lab, VirusTotal a další [27]).

Důvodem, proč se věnovat oboru ochrany před malwarem, je prospěšnost a perspektivnost tohoto oboru. Není reálné, že by v nejbližších letech zanikla potřeba se před škodlivým softwarem chránit. Naopak, je (a také v budoucnu bude) třeba velice rychle reagovat na nové typy malwaru a bránit jejich šíření.

Tato technická dokumentace k bakalářské práci se skládá ze šesti kapitol — tohoto úvodu, čtyř obsahových kapitol a závěru. Ve druhé kapitole bude představen formát souborů PE. V kapitole č. 3 bude nastíněna teorie týkající se analýzy binárních souborů. Dále bude ukázáno, jakým způsobem je řešena analýza malwaru ve společnosti Avast Software. Důležitou částí bude představení nástroje YaraGen, který sloužil jako základ této práce. Ve čtvrté kapitole budou navrženy nové metody a postupy pro automatickou detekci škodlivého kódu. V páté kapitole pak bude ukázáno, jakým způsobem byly tyto nové postupy implementovány.

# Kapitola 2

# Formát spustitelných souborů PE

Formát *Portable Executable* (PE) je formát souboru používaný v prostředí operačních systémů Microsoft Windows. Důvod, proč se zaměřit na systémy Microsoft Windows, je prostý. Podle nejnovějších údajů [24] jsou tyto systémy používány na více než 75 % osobních počítačů na celém světě. Na tento systém je zaměřeno nadpoloviční množství veškerého malwaru (67 % na přelomu let 2017 a 2018 [1]). Tento podíl v posledních letech klesá, v oblasti osobních počítačů je však stále velká většina malwaru zaměřena právě na Windows.

Formát PE se používá pro spustitelné soubory, dynamicky připojované knihovny, objektové soubory a další typy souborů. Nejčastěji se s ním můžeme setkat ve formě souborů s příponou *.exe* či *.dll*. Informace v této kapitole budou vycházet především z oficiální dokumentace formátu PE [15]. Některé údaje budou vycházet z diplomové práce Ing. Marka Milkoviče [18], která se formátu PE blíže věnuje.

První dva bajty PE souboru tvoří takzvané magické číslo. Jedná se o hodnotu 0x5A4B, což jsou ASCII hodnoty znaků 'M' a 'Z' (podle iniciálů tvůrce formátu PE Marka Zbikowského). Poté následuje DOS hlavička, tzv. DOS pahýl (DOS *stub*) a PE hlavička. Za nimi se nachází tabulka sekcí a poté jednotlivé sekce (například *.data*, *.text*).

DOS pahýl je program, který v případě spuštění v operačním systému MS-DOS vypíše zprávu *„This program cannot be run in DOS mode“*. DOS pahýl a DOS hlavička se v souborech formátu PE nacházejí pouze z důvodu zpětné kompatibility a z pohledu této práce nejsou důležité.

Mnohem podstatnější je PE hlavička, která obsahuje informace důležité pro spouštění programu. Nachází se zde údaje o typu cílové architektury, ukazatel na tabulku symbolů, adresa vstupního bodu programu a podobně. Nachází se zde i datové složky, které budou blíže popsány v následující podkapitole.

## 2.1 Datové složky

Struktura datových složek (dále nazývány zavedenějším anglickým termínem *data directories*) v sobě obsahuje 16 položek odkazujících na dodatečné tabulky. Každá tato položka se skládá z adresy, na které tabulka začíná, a velikosti tabulky. Tři z pohledu této práce nejdůležitější záznamy v *data directories* budou nyní blíže představeny.

### 2.1.1 Tabulka importovaných symbolů

Běžně se stává, že spustitelný soubor využívá funkce či symboly třetích stran. Tabulka importovaných symbolů se obvykle nachází v sekci *.idata* a obsahuje informace důležité

pro načítání těchto symbolů. Symboly se importují z knihoven (typicky soubory s příponou *.dll*) — může se jednat jak o systémové, tak uživatelské knihovny. Při načítání souboru je nutné tyto knihovny načíst do adresového prostoru spouštěného procesu.

Každá položka tabulky importovaných symbolů odpovídá jedné knihovně, ze které soubor importuje symboly. V každé této položce se nacházejí dvě další tabulky — tzv. *Import lookup table* (ILT) a *Import address table* (IAT). V ILT se nacházejí záznamy popisující jednotlivé symboly, které se z knihovny importují. Pro správné načtení symbolů potřebujeme znát jejich relativní adresu v rámci knihovny. V různých verzích knihovny se však může nacházet ten samý symbol pokaždé na jiné adrese. Proto v ILT nemůže být napevno nastavena adresa symbolu, ale nachází se tam pouze index do tabulky exportů dané knihovny nebo jméno importovaného symbolu.

Při načítání souboru se vyhledá symbol podle indexu uvedeného v ILT a jeho adresa se zapíše do IAT. Při běhu programu se pak používají právě adresy z IAT.

### 2.1.2 Tabulka zdrojů

Zdroje (dále bude používán zavedenější anglický termín *resource*) jsou data sloužící pouze ke čtení (*read-only*), která jsou vložena do souboru formátu PE. Obvykle se nacházejí v sekci *.rsrc*. Může se jednat o jakákoliv data, časté je však použití *resource* například pro ikonu souboru, řetězce používané pro internacionalizaci a lokalizaci či informace o verzi. Přehled všech předdefinovaných typů *resource* je uveden v tabulce 2.1. Uživatel (programátor) si také může nadefinovat vlastní typy.

Uložení *resource* v souboru je ve formě stromu. Většinou má tento strom tři úrovně. Na nejvyšší úrovni se nachází tabulka, která odkazuje na uzly druhé úrovně. Každý uzel druhé úrovně odpovídá jednomu typu *resource* a odkazuje na uzly třetí úrovně, které už obsahují samotná data.

#### Uložení ikon

Nejvyšší pozornost ze všech typů *resource* bude v této práci věnována ikonám, proto je vhodné uvést pár slov ohledně uložení ikon v *resource*.

Pro vkládání ikon do souboru formátu PE se používají soubory s příponou *.ico*. Do jednoho takového souboru můžeme umístit více ikon. Zároveň můžeme do jednoho souboru PE nahrát ikony z více souborů formátu *ico*. Proto na rozdíl od většiny jiných *resource* nenáleží ikonám pouze jeden typ *resource*, ale rovnou dva — `RT_GROUP_ICON` a `RT_ICON`. Záznam `RT_GROUP_ICON` odpovídá jednomu souboru typu *ico* a obsahuje odkazy na všechny ikony, které byly z tohoto souboru získány. Záznam `RT_ICON` pak obsahuje samotná data ikony.

### 2.1.3 Tabulka certifikátů

Certifikáty se používají k tzv. podepisování souborů. Smyslem takového podepisování je ověření, že soubor pochází z důvěryhodného zdroje a nebyl nijak změněn jeho obsah. Ze souboru je vypočítán hash, který je podepsán certifikační autoritou. Podepsaná podoba je poté vložena do obsahu certifikátu a certifikát je vložen do tabulky certifikátů. Při spouštění souboru pak systém Windows vypočítá hash a zkontroluje, zda odpovídá hodnotě uložené v certifikátu. Tak je zajištěno, že obsah souboru je prakticky nemožné změnit, aniž by to bylo systémem detekováno. Jak však ukazuje článek z roku 2017 [11], v dnešní době se stále

| Název | Hodnota | Popis |
|---|---|---|
| RT_CURSOR | 0x1 | Kurzor |
| RT_BITMAP | 0x2 | Bitmapový obrázek |
| RT_ICON | 0x3 | Ikona |
| RT_MENU | 0x4 | Menu |
| RT_DIALOG | 0x5 | Dialogový box |
| RT_STRING | 0x6 | Řetězec |
| RT_FONTDIR | 0x7 | Tabulka fontů |
| RT_FONT | 0x8 | Font písma |
| RT_ACCELERATOR | 0x9 | Klávesa či klávesová zkratka s daným chováním |
| RT_RCDATA | 0xA | Aplikačně specifická data |
| RT_MESSAGETABLE | 0xB | Tabulka zpráv |
| RT_GROUP_CURSOR | 0xC | Skupina záznamů RT_CURSOR |
| RT_GROUP_ICON | 0xE | Skupina záznamů RT_ICON |
| RT_VERSION | 0x10 | Informace o verzi |
| RT_DLGINCLUDE | 0x11 | Jméno hlavičkového souboru, který obsahuje symbolická jména *resource* |
| RT_PLUGPLAY | 0x13 | Plug and play |
| RT_VXD | 0x14 | *VXD driver* |
| RT_ANICURSOR | 0x15 | Animovaný kurzor |
| RT_ANIICON | 0x16 | Animovaná ikona |
| RT_HTML | 0x17 | HTML |
| RT_MANIFEST | 0x18 | Manifest |

Tabulka 2.1: Předdefinované typy *resource*

častěji setkáváme s podepsaným malwarem, který obsahuje například certifikáty odcizené jiným aplikacím.

Pokud je soubor formátu PE podepsaný, obsahuje v tabulce certifikátů jeden nebo více certifikátů. Pro účely analýzy je důležitý pouze první z nich, ostatní jsou certifikáty vyšších certifikačních autorit v rámci tzv. řetězce důvěry.

Formát, který se pro digitální podpis používá, se nazývá *Microsoft Authenticode* [16, 17]. Certifikáty jsou uloženy ve struktuře *Signed data* podle standardu PKCS #7 [10], která má formát specifikovaný pomocí abstraktní syntaktické notace ASN.1 [25] a je kódována pomocí kódování DER [26]. Ve struktuře PKCS #7 se kromě certifikátů ve formátu X.509 [4] nachází také hash souboru, informace o podepisujícím, případně další údaje, jako například časové razítko. Detailnější popis, který je nad rámec této práce, je možné nalézt v dokumentu společnosti Microsoft [17].

## 2.2 Sekce

Za *data directories* se nachází tabulka sekcí. Sekce je část souboru, která uchovává data stejného typu. Již dříve byla zmíněna sekce *.idata* obsahující tabulku importovaných symbolů. Další běžně používané sekce jsou uvedeny v tabulce 2.2. Není nutné dodržovat pojmenování sekcí, ale linker obvykle při sestavování programu používá právě tuto konvenci jmen.

| Název | Popis |
|---|---|
| *.text* | Kód programu |
| *.data* | Inicializovaná data |
| *.bss* | Neinicializovaná data |
| *.rdata* | Konstantní (*read-only*) data |
| *.idata* | Tabulka importovaných symbolů |
| *.edata* | Tabulka exportovaných symbolů |
| *.rsrc* | *Resource* |
| *.reloc* | Relokační tabulka |

Tabulka 2.2: Nejběžnější jména sekcí

Sekce jsou zarovnány na stránky paměti, takže je možné každé sekci nastavit různé příznaky. Kupříkladu stránkám paměti obsahujícím sekci *.text* s kódem programu se nastaví příznak *read-only*, který nedovolí modifikaci dat v této sekci.

Každý záznam v tabulce sekcí obsahuje několik hodnot, z nejzajímavější je jméno sekce, adresa začátku sekce, velikost sekce a charakteristika sekce.

# Kapitola 3

# Analýza škodlivého kódu

Tato kapitola obsahuje shrnutí technik a nástrojů používaných pro detekci malwaru. Jedná se většinou o obecné postupy. V některých případech bude popis zaměřen konkrétně na technologie společnosti Avast Software a toto bude v textu výslovně zmíněno.

Nejprve se seznámíme s analýzou binárních souborů. Budou ukázány možnosti, jak k této analýze přistupovat. Dále bude představen program YARA, který je pro celou tuto práci naprosto zásadní. Také budou popsány prostředky používané ve společnosti Avast Software, které jsou nezbytné pro fungování nástroje YaraGen. Nakonec, po předložení všech relevantních faktů, bude představen samotný systém YaraGen, který byl v rámci tento práce doplněn o nové funkce a možnosti.

## 3.1 Rozdělení analýz binárních souborů

V této části budou představeny dva základní přístupy, které se používají pro analýzu binárních souborů, tedy i malwaru. Jedná se o statickou a dynamickou analýzu. Budou zmíněny výhody i nevýhody obou typů a bude ukázáno, proč je ideální oba přístupy kombinovat.

### 3.1.1 Statická analýza

Statická analýza je prvním ze dvou zásadních typů analýz binárních souborů. Spočívá v rozboru struktury a určitých zajímavých částí samotného binárního souboru. Soubor je tedy analyzován, aniž by byl spouštěn. To s sebou nese určité výhody. Především je tento přístup naprosto bezpečný. Pokud binární soubor (potenciální malware) není spuštěn, nehrozí žádné nebezpečí napadení počítače či ztráty dat. Také bývá tato analýza rychlejší. Je tedy často používána pro prvotní seznámení se souborem. Může nám přinést mnoho zajímavých informací. Například, pokud jsou v souboru odhaleny stejné rysy, které obsahuje nějaká skupina malwaru, je velice dobře možné, že analyzovaný soubor patří také do této skupiny malwaru.

Statická analýza má však také svoje nevýhody. První spočívá v tom, že je nutné velmi pečlivě vybrat, na které statické vlastnosti se zaměřovat. Některé rysy binárních souborů se objevují v malwaru, ale vyskytují se i v neškodném softwaru. Pokud by byly klasifikovány soubory jako škodlivé pouze na základě takového rysu, docházelo by k mnoha chybám ve statistice označovaným jako chyby prvního typu (anglicky *false positive*).

Další nevýhodou statické analýzy je to, že nedokáže postihnout všechny důležité vlastnosti. Některé informace, jako například údaje o síťové komunikaci daného souboru, se dají získat pouze tak, že je binární soubor spuštěn.

Třetí nevýhoda vychází z toho, že tvůrci malwaru mohou použít specializované nástroje, např. takzvané *packery*, které statickou analýzu komplikují. Princip *packerů* spočívá v tom, že binární soubor je zkomprimován. Je k němu také přidán kód, který dokáže zkomprimovaný archiv dekomprimovat. Při spuštění nejprve tento úsek kódu soubor dekomprimuje a poté je již spuštěn samotný škodlivý kód. V angličtině se pro tento přístup používá zkratka SEA (*self-extracting archive*). Problém *packerů* se dá částečně řešit pomocí programů, které umí kód dekomprimovat — *unpackerů*.

Komprimování binárních souborů je možné zařadit do množiny technik souhrnně nazývaných obfuskace. Patří sem různé způsoby zatemnění kódu tak, aby vykazoval požadované chování, ale pro člověka byl velmi těžko čitelný. Obfuskace obecně statickou analýzu neznemožňuje, ale může ji zkomplikovat.

Za formu statické analýzy lze považovat i dvě techniky reverzního inženýrství — disassemblování a dekompilace. *Disassembler* rekonstruuje z binárního kódu instrukce v jazyku symbolických adres. Dekompilátor rekonstruuje z binárního kódu zdrojový kód v nějakém vyšším programovacím jazyce (C++, Java). *Disassemblery* a dekompilátory nejsou náplní této práce, proto nebudou podrobněji popisovány.

I přes zmíněné nevýhody má statická analýza své nezastupitelné místo v analýze binárních souborů. Konkrétní rysy, na které se zaměřuje statická analýza v nástroji YaraGen, budou podrobně popsány v kapitole 3.3.2.

### 3.1.2 Dynamická analýza

Druhým typem analýz binárních souborů jsou analýzy dynamické. Na rozdíl od statických analýz zde dochází ke spouštění kódu obsaženého v binárním souboru. Jak již bylo naznačeno v minulé kapitole, tento přístup je potenciálně nebezpečný, pokud není proveden s rozmyslem. Nebylo by příliš moudré spouštět škodlivý kód běžně na počítači a čekat, co provede. Proto se využívá specializovaných programů, tzv. sandboxů (anglický výraz sandbox se používá i v českém názvosloví). Proces spuštěný v sandboxu dostane pouze omezené prostředky, které byly předem definovány. Je možné sledovat běh binárního souboru, ale zároveň nehrozí žádné nebezpečí napadení počítače či znehodnocení dat. Analýza malwaru je bez sandboxů v dnešní době prakticky nepředstavitelná.

Samotná dynamická analýza se pak typicky skládá ze dvou částí. Nejprve je binární soubor spuštěn v sandboxu a je sledováno a zaznamenáváno jeho chování. Výsledkem této fáze je nějaký soubor (zpráva, anglicky *report*), který toto chování popisuje. Běžné sandboxy obecně žádné takové soubory neprodukují. Proto se používají speciální sandboxy pro analýzu malwaru (např. Cuckoo [5]), které takovéto zprávy o chování binárního souboru umějí poskytnout. Formát těchto souborů může být různý, používá se například XML či JSON.

Ve druhé fázi je již prováděna samotná analýza. Analyzován však není původní binární soubor, nýbrž zpráva o jeho běhu, která byla vygenerována sandboxem. Je možné si všimnout, že pokud sandbox budeme považovat za jakousi černou skříňku, které předložíme vstup (binární soubor) a ona vrátí výstup (zpráva o chování binárního souboru), začne se samotná dynamická analýza principiálně blížit statické analýze. Tedy i dynamická analýza je řešena analyzováním nějakého statického souboru. Rozdílem je pouze to, že u dynamických analýz je tímto statickým souborem výstup sandboxu.

Nevýhodou dynamických analýz je nižší rychlost. Časově náročnou operací je především spuštění souboru v sandboxu. Je tedy například možné soubor analyzovat staticky a až poté v případě potřeby i dynamicky.

Nevýhoda nižší rychlosti analýzy je však bohatě vyvážena užitkem, který nám tato analýza přináší. Je to totiž jediný způsob, jak získat některé důležité údaje, jako například informaci o tom, do kterých souborů program zapisuje, které webové stránky načítá, apod. Je také nutné zmínit, že dynamické (behaviorální) rysy jsou obecně považovány za důvěryhodnější než statické. Statické rysy se totiž dají zatemňovat (obfuskovat), což je u dynamických velmi obtížné. Pokud se binární soubor chová nebezpečným způsobem, v sandboxu je toto chování odhaleno, i když je soubor obfuskován či například zkomprimován pomocí *packeru*. Navíc je u analýzy dynamických rysů obecně menší riziko prohlášení neškodného souboru za malware (chyba prvního typu, *false positive*). Když totiž neškodný soubor obsahuje např. podobné importované symboly či bajty na vstupní bodu programu, jako nějaký známý malware, statická analýza nemá žádné další prostředky, jak soubor od malwaru odlišit. Při dynamické analýze však pravda často vyjde najevo.

V posledních letech se začíná dostávat do popředí malware typu *Sandbox evasion* [28] (např. *ransomware* Locky). Tento malware dokáže speciálními technikami detekovat, že je spuštěn v sandboxu a v takové situaci se chová neškodně, aby nebylo odhaleno jeho pravé chování. Při boji proti tomuto zákeřnému typu škodlivého softwaru je však třeba reagovat na úrovni sandboxu, nikoliv na úrovni analýzy výstupů sandboxu. Z tohoto důvodu je technika *Sandbox evasion* mimo oblast zájmu této práce a je uvedena pouze jako zajímavý příklad problémů, kterým musí dynamická analýza čelit.

## 3.2 Nástroje pro analýzu malwaru v Avast Software

Tato podkapitola pojednává o třech nejdůležitějších nástrojích, které jsou relevantní pro systém YaraGen a používají se ve společnosti Avast Software. YaraGen je se všemi těmito nástroji velice těsně svázán a bez nich by nemohl v současné verzi fungovat.

### 3.2.1 YARA

YARA je open-source nástroj, jehož cílem je pomáhat analytikům identifikovat a klasifikovat malware. Na svých oficiálních stránkách [27] je popisován jako švýcarský nůž pro vyhledávání vzorů pro analýzu malwaru. Na těchto stránkách se také dozvíme, že je široce používán napříč mnoha společnostmi v oboru boje proti malwaru. YARA byla vytvořena v rámci společnosti VirusTotal a jejím tvůrcem je Victor Alvarez. Akronym YARA je zkratkou slov *Yet Another Recursive Acronym* či *Yet Another Ridiculous Acronym*.

Systém YARA se skládá ze dvou částí. Z deklarativního jazyka YARA umožňujícího popisovat vzory v souborech, a stejnojmenného programu vyhledávajícího tyto vzory v souborech. Důležitou roli hrají také takzvané moduly, které umožňují specifikovat komplexnější chování.

#### Jazyk YARA

Jazyk YARA je používán k popisů vzorů v souborech. Nemusí se přitom jednat jenom o binární soubory, i když typické je použití právě pro tento typ souborů. Jazyk YARA definuje syntaxi a sémantiku souborů obsahujících popis hledaných vzorů. Tyto soubory v jazyce YARA mají obvykle příponu *.yara* či *.yar*, i když to není podmínkou. Typický YARA soubor se skládá z následujících částí:

- (importování modulů)

- YARA pravidla
- (komentáře)

Části v závorkách jsou nepovinné. Pokud není použit žádný modul (o modulech více v kapitole 3.2.1), není samozřejmě nutné ani žádný modul importovat. Komentáře jsou pak také pochopitelně volitelné. YARA pravidla tvoří hlavní obsah YARA souboru a i když nejsou povinná, vytvářet YARA soubor bez jediného pravidla by nemělo žádný smysl.

Syntaxe jazyka YARA vychází z programovacího jazyka C (i když YARA není programovacím jazykem). V následující tabulce jsou uvedena klíčová slova jazyka YARA, která nemohou být užita jako identifikátory.

| | | | | | | |
|---|---|---|---|---|---|---|
| `all` | `and` | `any` | `ascii` | `at` | `condition` | `contains` |
| `entrypoint` | `false` | `filesize` | `fullword` | `for` | `global` | `in` |
| `import` | `include` | `int8` | `int16` | `int32` | `int8be` | `int16be` |
| `int32be` | `matches` | `meta` | `nocase` | `not` | `or` | `of` |
| `private` | `rule` | `strings` | `them` | `true` | `uint8` | `uint16` |
| `uint32` | `uint8be` | `uint16be` | `uint32be` | `wide` | `xor` | |

Tabulka 3.1: Klíčová slova jazyka YARA

Hlavní jednotkou YARA souboru je pravidlo. Jeden YARA soubor může obsahovat jedno, nebo i více pravidel. Každé pravidlo je uvozeno klíčovým slovem *rule* následovaným jménem pravidla.

```
rule dummy
{
    condition:
      false
}
```

Kód 3.1: Nejjednodušší pravidlo převzaté z oficiální dokumentace [29]

Pravidla se skládají z následujících tří částí:

- (metainformace)
- (řetězce)
- podmínky

Metainformace jsou nepovinnou položkou. Na vyhodnocování pravidla nemají žádný vliv, slouží pouze jako informace pro uživatele. Metainformace jsou uvozeny klíčovým slovem *meta* a jsou ve tvaru *identifikátor: hodnota*. Hodnota může být řetězec, celé číslo nebo pravdivostní hodnota (*true* či *false*). Příklad použití metainformací je v kódu 3.2.

Řetězce taktéž nejsou povinnou součástí pravidla. Definice každého řetězce se skládá z unikátního identifikátoru a samotného řetězce. Identifikátor musí začínat znakem *$*, za kterým následuje sekvence alfanumerický znaků a podtržítek.

Samotný řetězec může být vyjádřen třemi různými formáty, jak je vidět v ukázce 3.2. První je způsob známý např. z jazyka C, tedy řetězec v uvozovkách. V tomto případě je ještě možné rozlišit ASCII řetězce a široké řetězce (dva bajty na znak). Také je možné pomocí

klíčového slova *nocase* vynutit vyhledávání bez ohledu na velikost písmen (*case insensitive*). Druhým formátem řetězců je sekvence hexadecimálně zapsaných bajtů ve složených závorkách. Třetí možností je použít regulární výrazy se syntaxí převzatou z jazyka Perl.

```
rule strings_example
{
    meta:
       string_meta = "String"
       int_meta = 42
       bool_meta = false
    strings:
       $ascii_str = "123"
       $hex_str = { 01 02 03 }
       $wide_str = "another example" wide
       $ascii_or_wide = "yet another example" ascii wide
       $regex = /state: (on|off)/
    condition:
       all of them
}
```

Kód 3.2: Příklad použití řetězců. Pozor na rozdílnost prvních dvou řetězců. Zatímco *$ascii_str* označuje znaky s hodnotami 0x31, 0x32 a 0x33 (ASCII hodnoty znaků '1', '2' a '3'), *$hex_str* popisuje řetězec znaků s hodnotami 0x01, 0x02 a 0x03.

V hexadecimálních řetězcích je možné používat další speciální konstrukce, které umožňují větší flexibilitu. Jedná se o tyto tři konstrukce:

- ? — zástupný znak (anglicky *wildcard*), zastupuje libovolnou hodnotu půlbajtu
- [M–N] — skok, reprezentuje M–N libovolných bajtů ($M \leq N$)
- X|Y — alternativa, reprezentuje dvě různé varianty hodnot

```
rule hex_strings_example
{
    strings:
       $hex_str1 = { 12 ?3 }
       $hex_str2 = { 00 [1--3] 11 }
       $hex_str3 = { F4 ( 62 B4 | 56 ) 45 } // F4 62 B4 45 or F4 56 45
    condition:
       all of them
}
```

Kód 3.3: Příklad speciálních konstrukcí v hexadecimálních řetězcích

Třetí část pravidla je povinná. Tato část obsahuje podmínky. Podmínky se řetězí běžným způsobem známým z programovacích jazyků pomocí operátorů *and*, *or*, případně *xor*. Při analýze nástrojem YARA (viz následující podkapitola) je podmínka vyhodnocena vůči vstupnímu souboru. Pokud je vyhodnocena jako pravdivá, znamená to, že dané pravidlo má shodu na daném souboru.

### Nástroj YARA

Nástroj YARA je program používaný pro vyhledávání vzorů v souborech. Vzory, které nástroj YARA vyhledává, jsou zapsány ve formě YARA pravidel. Vstupem programu je množina souborů, které mají být prohledány, a jeden soubor s YARA pravidly. Výstupem je pro každý vstupní soubor množina pravidel, pro která byla splněna podmínka.

```
rule example_rule
{
    meta:
       author = "Name of author"
       date = "1.1.2019"
    strings:
       $hex_str = { 12 ?3 }
       $ascii_str = "Example"
       $regex = /\d{1,2}\.\d{1,2}\.\d{4}/
    condition:
       ($hex_str and $ascii_str) or #regex > 2
}
```

Kód 3.4: Příklad pravidla obsahujícího metainformace, řetězce i složitější podmínku. Pravidlo bude splněno, pokud analyzovaný binární soubor bude obsahovat $hex_str a současně $ascii_str, nebo pokud bude nejméně dvakrát obsahovat $regex.

### Moduly YARA

Moduly jsou způsobem, kterým je YARA rozšiřována o další užitečné funkce. S jejich pomocí je možné v YARA pravidlech definovat datové struktury a funkce použitelné k vytváření komplexnějších pravidel. Jejich významným přínosem je přidání sémantiky zpracovávaným datům. Je tak vytvořena určitá vrstva abstrakce, kdy se na analyzovaný soubor nenahlíží už jen jako na posloupnost bajtů, ale například jako na spustitelný soubor typu Portable Executable (PE), který obsahuje určité hlavičky, sekce, atd. Teprve použitím modulů se YARA stává opravdu užitečným nástrojem pro analýzu malwaru.

Podle dokumentace [29] obsahuje YARA ve verzi 3.10.0 tyto moduly:

- *pe* — Funkce na práci se soubory formátu Portable Executable (PE)
- *elf* — Podobný jako modul pe, ale pro soubory formátu Executable and Linkable Format (ELF)
- *cuckoo* — Parsování zprávy ze sandboxu Cuckoo ve formátu JSON
- *magic* — Umožňuje zjistit typ souboru
- *hash* — Podpora pro počítání hashů (např. MD5, SHA256)
- *math* — Matematické funkce (např. střední hodnota, směrodatná odchylka, entropie)
- *dotnet* — Parsování .NET souborů
- *time* — Obsahuje pouze funkci *now()*, která vrací aktuální čas

Pro účely této práce budou nejdůležitější moduly *pe* a *cuckoo*.

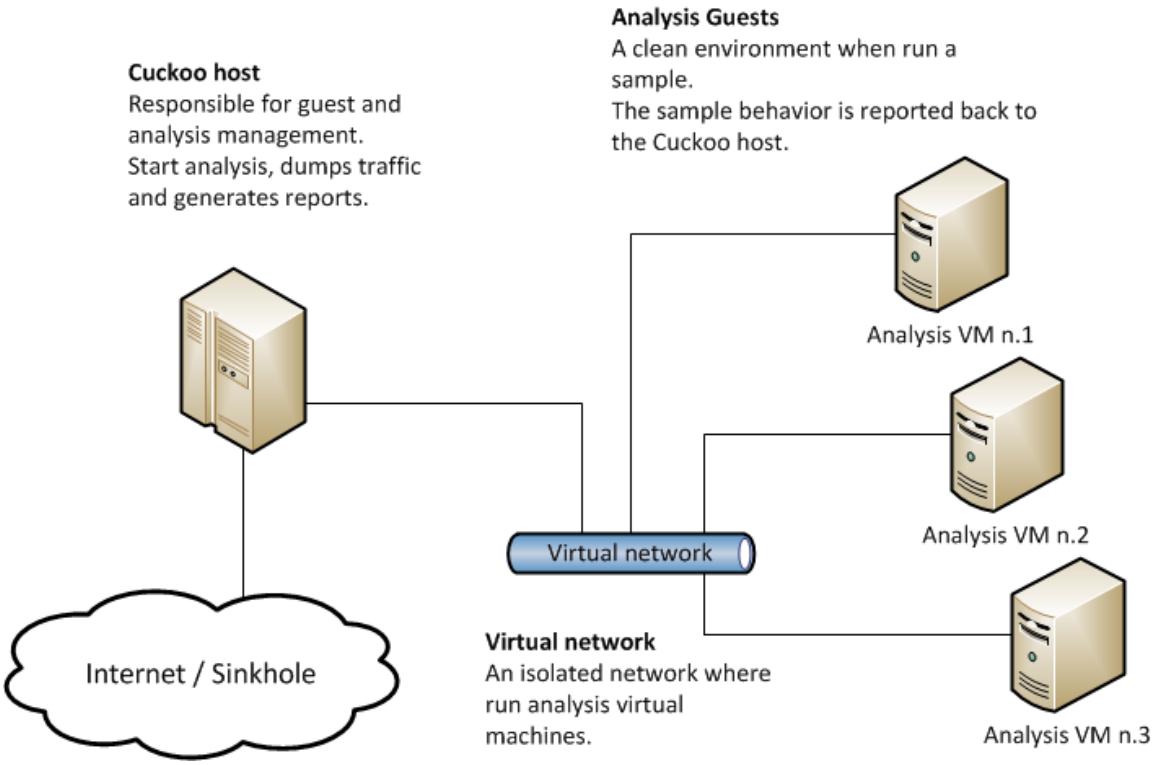
### 3.2.2 Fileinfo

Fileinfo [23] je nástroj pro statickou analýzu binárních souborů. Jedná se o nástroj vyvinutý přímo v rámci společnosti Avast Software jako součást systému RetDec, který je nyní zveřejněn jako open-source [23]. Fileinfo dokáže analyzovat binární soubor (formátu PE, ELF, COFF, Mach-O či Intel HEX) a vyextrahovat z něj podstatné informace. Výstupem nástroje je soubor formátu JSON obsahující tyto vyextrahované informace. Nad tímto souborem poté mohou být prováděny další analýzy.

Výhodou použití nástroje Fileinfo je to, že při analýze binárních souborů různých typů je možno pracovat s jednotným formátem souboru JSON. Další nezanedbatelnou výhodou je jistá úroveň abstrakce — není nutné parsovat samotný binární soubor, ale stačí parsovat soubor ve formátu JSON, což je o poznání jednodušší a pohodlnější.

### 3.2.3 Cuckoo

Dalším nástrojem relevantním pro tuto práci je open-source sandbox pro analýzu malwaru zvaný Cuckoo. Jak již bylo zmíněno v kapitole 3.1.2, sandboxy hrají velice důležitou roli v dynamické analýze binárních souborů.




Obrázek 3.1: Obrázek převzatý z oficiální dokumentace [5] reprezentuje architekturu systému Cuckoo

Cuckoo provádí dynamickou analýzu mnoha různých typů souborů — spustitelných souborů, DLL souborů, PDF dokumentů, Visual Basic skriptů, atd. Základní architektura systému Cuckoo je ukázána na obrázku 3.1. Skládá se z hostitele (*host*) a hostů (*guests*). Hostitel je řídicí software, který zajišťuje řízení analýzy, delegování úkolů na hosty a genero-

vání zpráv ve formátu JSON. Hosté jsou potom fyzická nebo virtuální zařízení, na kterých probíhá v bezpečném a izolovaném prostředí spouštění souborů a analýza jejich chování.

Zpráva vygenerovaná nástrojem Cuckoo obsahuje mnoho informací. Mezi informace nejpodstatnější pro tuto práci patří jména pojmenovaných objektů (mutexy, semafory, ...), které analyzovaný soubor vytvořil. Dále jsou důležité informace o síťové aktivitě nebo práci se souborovým systémem či registrovými klíči.

## 3.3 YaraGen

YaraGen je nástroj vytvořený Ing. Markem Milkovičem v rámci jeho diplomové práce [18]. Jedná se o nástroj pro automatické generování vzorů z množiny binárních souborů. Ve verzi vytvořené Ing. Milkovičem se jedná o nástroj pro statickou analýzu, který je však implementován modulárně a nabízí tak možnosti snadného rozšíření o další analýzy.

### 3.3.1 Princip činnosti

Práce nástroje YaraGen spočívá v tom, že ze vstupních souborů vyextrahuje podstatné informace a vytvoří detekční vzor ve formátu pravidla jazyka YARA. Pracuje s pojmem *skupina pokrytí*, který definuje, že určitá hodnota se musí objevit minimálně v jistém množství souborů, aby byla zahrnuta do výsledného YARA pravidla. Kolik souborů musí hodnota pokrývat, aby byla zahrnuta do výsledného vzoru, určují konfigurace pro danou analýzu (seznam podporovaných analýz viz níže v této kapitole). Tyto konfigurace také určují restriktivnost analýz, tedy jestli jsou hodnoty ve výsledném pravidle spojeny logickými spojkami *and*, nebo *or*.

Pro správné fungování je třeba YaraGen použít ve spolupráci s nástrojem Fileinfo (kapitola 3.2.2). Z důvodu modularity a znovupoužitelnosti kódu je statická analýza rozdělena právě mezi tyto dva nástroje. Pomocí nástroje Fileinfo je nejprve zanalyzován každý binární soubor zvlášť. Výsledkem této analýzy je množina vlastností vyextrahovaná ze vstupního souboru, která je uložena do souboru ve formátu JSON. Tedy pro každý analyzovaný binární soubor je získána jedna zpráva formátu JSON. Na vstupu nástroje YaraGen je množina takovýchto souborů formátu JSON.

YaraGen tedy vůbec nepracuje s analyzovanými binárními soubory. Neprovádí žádné nízkoúrovňové činnosti, jako například získávání údajů z hlaviček binárních souborů. Zaměřuje se pouze na vysokoúrovňové zpracování zpráv formátu JSON vygenerovaných pomocí nástroje Fileinfo.

Toto rozdělení analýzy mezi dva nástroje přináší výhody. Kromě lepší udržovatelnosti zdrojového kódu se jedná také o zefektivnění analýzy. Je například možné nejprve vytvořit zprávy z nástroje Fileinfo a nad těmito zprávami potom opakovaně provádět analýzy s různými konfiguracemi. Vzhledem k tomu, že analýza souboru nástrojem Fileinfo je poměrně časově náročná operace, může tento přístup přinést znatelnou časovou úsporu.

### 3.3.2 Seznam podporovaných analýz

Nástroj YaraGen ve verzi vytvořené v rámci diplomové práce Ing. Milkoviče podporuje několik statických analýz. Každá analýza má svoji implicitní konfiguraci (minimální procentuální pokrytí, restriktivnost). Tuto konfiguraci je možné explicitně změnit použitím konfiguračního souboru. Většina analýz podporuje binární soubory typu Portable Execu-

table (PE), případně .NET soubory. Další informace k analýzám je možné nalézt ve zmíněné diplomové práci [18].

### Název .NET třídy

V systému YARA sice modul určený pro zpracování .NET souborů je, ale tento modul (v aktuální verzi YARA 3.10.0) nepodporuje vyhledávání názvů .NET tříd. Proto je nutné vyhledávat názvy .NET tříd jako běžné řetězce.

### Název .NET metody

Jedná se o stejnou situaci jako v případě .NET tříd. Problém se opět řeší vyhledáváním názvů .NET metod jako běžných řetězců.

### TypeLib identifikátor

Jedná se o hodnotu často obsaženou v .NET souborech. Je to globálně unikátní identifikátor (GUID) o velikosti 32 bajtů. Stejně jako v předchozích případech neobsahuje YARA podporu pro vyhledávání TypeLib identifikátoru, takže je nutné jej vyhledávat jako řetězec.

### Řetězce

Nejproblematičtějším bodem analýzy řetězců je jejich identifikace. Při analýze binárního souboru totiž není možné spolehlivě určit, která sekvence bajtů je řetězcem a která není. Proto obsahuje nástroj Fileinfo heuristiky, pomocí kterých vyhledává sekvence, které by pravděpodobně měly být řetězci. Nástroj YaraGen pak už pouze zpracovává tyto nalezené sekvence.

### Importované symboly

V tomto případě je možno využít YARA modul pro zpracování PE souborů, konkrétně funkci *pe.imports()*. Prvním argumentem funkce je jméno knihovny (DLL), druhým je jméno funkce.

```
import "pe"
rule imports_example
{
    condition:
        pe.imports("kernel32.dll", "WriteProcessMemory")
}
```

Kód 3.5: Příklad pravidla obsahujícího importované symboly.

### Vstupní bod

Vstupní bod (*entry point*) je místo v kódu, kde se začne provádět první instrukce programu. U souborů typu Portable Executable (PE) je možné adresu vstupního bodu získat z PE hlavičky. Samotná adresa vstupního bodu je však příliš slabá informace. Proto je vhodné přidat ještě informaci o prvních bajtech, které se nacházejí za vstupním bodem. Využívá se dvojice podmínek v konjunkci, z nichž první definuje adresu vstupního bodu a druhá sekvenci bajtů na této adrese. Opět je použit modul *pe*, konkrétně atribut *pe.entry_point*.

(V jazyce YARA existuje také klíčové slovo *entrypoint*, jehož použití však od verze 3.0 není doporučeno a bude v budoucnosti z jazyka YARA kompletně vyřazeno.)

```
import "pe"
rule entry_point_example
{
    strings:
       $entry_point_bytes = {00 11 22 33 44 55}
    condition:
       pe.entry_point == 0x1234 and $entry_point_bytes at pe.entry_point
}
```

Kód 3.6: Příklad pravidla obsahujícího vstupní bod programu.

### Sekce

Typický program se dělí na několik logických částí zvaných sekce. Běžné jsou například sekce *.data* obsahující kód či *.bss* obsahující neinicializovaná data. Binární soubor lze tedy analyzovat na základě počtu a pořadí těchto sekcí. V jazyce YARA se k tomuto účelu používá atribut *pe.sections*, který obsahuje pole sekcí.

```
import "pe"
rule sections_example
{
    condition:
       pe.sections[0].name == ".text" and pe.sections[1].name == ".rdata"
}
```

Kód 3.7: Příklad pravidla obsahujícího sekce.

### Cesta k PDB souboru

Soubor PDB je soubor, který obsahuje ladící informace pro daný binární soubor typu PE či .NET. Umístění PDB souboru je určeno v PE hlavičce. Přesná shoda cesty PDB souboru je poměrně silná unikátní informace. YARA však nenabízí žádnou funkci na detekci cesty k PDB souboru, proto je nutné tuto cestu vyhledávat jako běžný řetězec.

### Rich hlavička

Rich hlavička (anglicky *rich header*) obsahuje informace o verzi překladače a knihoven. Pro její vyhledání je možné v jazyce YARA použít funkci *pe.rich_signature*.

### Podepisující

Pokud je spustitelný soubor podepsán, je možné analyzovat použitý certifikát. K tomu v jazyce YARA slouží atribut *pe.signatures*. Tento atribut obsahuje pole podpisů, avšak obvykle má soubor pouze jeden podpis, takže nejběžnější je použití *pe.signatures*[0].

# Kapitola 4

# Návrh metod pro generování nových rysů binárních souborů

Cílem této práce bylo vylepšení generování vzorů pro detekci škodlivého kódu. Za tímto účelem byly navrženy metody pro generování nových rysů binárních souborů a také byly vylepšeny stávající metody generování. Jedná se jak o metody statické, tak i dynamické analýzy. Jak již bylo nastíněno, tato práce velmi úzce navazuje na nástroj YaraGen, proto byly tyto metody navrhovány s cílem integrování do zmíněného nástroje. Neméně zásadní bylo při návrhu také počítat s nutností kompatibility s jazykem YARA. Dalším důležitým faktorem byla použitelnost v praxi. Nově navržené metody byly vytvářeny s cílem generování co možná nejkvalitnějších detekčních vzorů, které budou zároveň spolehlivé a přispějí tak k užitečnosti celého nástroje YaraGen.

V následujících podkapitolách budou podrobně představeny všechny navržené typy analýz a také budou ukázána navrhovaná vylepšení stávajících metod. Analýzy se primárně týkají binárních souborů ve formátu PE (*Portable Executable*).

Při představení jednotlivých metod generování rysů bude dodrženo základní rozdělení představené v kapitole 3.1, tedy rozdělení na statické a dynamické analýzy.

## 4.1 Generování statických rysů

V kapitole 3.3 bylo ukázáno, že nástroj YaraGen ve verzi vytvořené Ing. Markem Milkovičem již podporoval celou řadu metod generování statických rysů z binárních souborů. Mojí úlohou tedy bylo především identifikování problematických míst návrhu a vylepšení stávajících metod.

### 4.1.1 Imphash

Jedním ze znaků binárního souborů, na který se můžeme zaměřit, jsou importované symboly. Údaje o importovaných symbolech se nalézají v souboru formátu PE v tabulce importovaných symbolů. V původní verzi YaraGen do detekčních vzorů generoval konjunkci jednotlivých importovaných symbolů, jak je možné vidět na následujícím příkladě.

```
import "pe"
rule imports_example
{
    condition:
       pe.imports("KERNEL32.dll", "ExitProcess") and
       pe.imports("KERNEL32.dll", "GetACP") and
       pe.imports("KERNEL32.dll", "GetProcAddress")
}
```

Kód 4.1: Pravidlo obsahující importované symboly.

Je možné si všimnout, že při vyšším množství importovaných symbolů se také odpovídajícím způsobem zvyšuje velikost vygenerovaného YARA pravidla, což není příliš žádoucí jev. Pravidlo se totiž stává méně přehledné a čitelné. Mezi další nevýhody patří vysoká paměťová náročnost při ukládání informací o importovaných symbolech v databázi a také obtížné porovnávání importovaných symbolů a vyhledávání v nich.

V roce 2014 představila americká společnost Mandiant nový způsob, jak detekovat binární soubory na základě jejich importovaných symbolů [13]. Jedná se o takzvaný *imphash* (zkratka ze slov *import hash*), který se rychle prosadil do pozice průmyslového standardu. Jak již název napovídá, jedná se hash počítaný z importovaných symbolů. Algoritmus výpočtu je následující:

1. Získání jména každé importované funkce na základě jejího číselného kódu
2. Převedení jmen knihoven (DLL) i funkcí na malá písmena (*lowercase*)
3. Odstranění přípony souboru z názvů importovaných modulů
4. Uložení řetězců vzniklých z názvu knihovny a funkce v seřazeném seznamu
5. Aplikování hashovací funkce MD5 [22] na tento seřazený seznam

Na další ukázce můžeme vidět, jak vypadá nová verze pravidla, pokud se místo importovaných symbolů zaměříme na *imphash*.

```
import "pe"
rule imphash_example
{
    condition:
       pe.imphash() == "b8bb385806b89680e13fc0cf24f4431e"
}
```

Kód 4.2: Pravidlo s použitím *imphashe*.

### 4.1.2 Vstupní bod programu

YaraGen v původní verzi již podporoval analýzu vstupního bodu programu. Kromě adresy vstupního bodu se analýza zaměřovala také na sekvenci bajtů, která se nachází na této adrese. Analýza bajtů na vstupním bodu programu může být velice přínosná, neboť může odhalit, co binární soubor provádí bezprostředně po spuštění. Navíc shodná sekvence bajtů na vstupním bodu je poměrně silná informace, která naznačuje podobnost souborů.

Konkrétně YaraGen generoval konjunkci mezi dvěma podmínkami. První obsahovala adresu vstupního bodu, druhá pak sekvenci bajtů, která se nachází na této adrese. Při použití v praxi se ukázalo, že tato metoda často přináší kvalitní výsledky, v některých případech však nebyla zdaleka ideální.

Problémem tohoto přístupu je totiž fakt, že podobné soubory mnohdy obsahují na vstupním bodu sekvence bajtů, které jsou sice podobné, ale nikoliv úplně stejné. Pokud byla YaraGenu dána na vstup množina takovýchto souborů, výsledkem bylo pravidlo jazyka YARA, které bylo velmi dlouhé a nepřehledné, jak je možné vidět na ukázce.

```
import "pe"
rule entry_point_example_old
{
    strings:
       $ep0 = { 60 E8 E6 19 00 00 8B 74 00 00 00 E8 08 61 68}
       $ep1 = { 60 E8 E6 19 00 00 8B 74 11 11 11 E8 08 61 68}
       $ep2 = { 60 E8 E6 19 00 00 8B 74 22 22 22 E8 08 61 68}
       $ep3 = { 60 E8 E6 19 00 00 8B 74 33 33 33 E8 08 61 68}
       $ep4 = { 60 E8 E6 19 00 00 8B 74 44 44 44 E8 08 61 68}
    condition:
       (pe.entry_point == 0x4000 and $ep0 at pe.entry_point) or
       (pe.entry_point == 0x4000 and $ep1 at pe.entry_point) or
       (pe.entry_point == 0x4000 and $ep2 at pe.entry_point) or
       (pe.entry_point == 0x4000 and $ep3 at pe.entry_point) or
       (pe.entry_point == 0x4000 and $ep4 at pe.entry_point)
}
```

Kód 4.3: Pravidlo vygenerované starou verzí YaraGenu ze souborů s podobnou sekvencí bajtů na vstupním bodu. Červenou barvou jsou zvýrazněny bajty, ve kterých se jednotlivé sekvence liší.

Dalším nedostatkem, kterého je možné si v ukázce povšimnout, je opakování stále stejné adresy vstupního bodu v každé jednotlivé podmínce v disjunkci.

Oba tyto nedostatky jsou nezanedbatelné, ale týkají se pouze přehlednosti a čitelnosti pravidla. Je zde ovšem i třetí problém, který se již týká použitelnosti pravidla. Od nástroje YaraGen je požadováno, aby jím vygenerovaná YARA pravidla dobře generalizovala, což toto pravidlo nesplňuje.

Předpokládejme, že pravidlo v ukázce bylo vygenerováno z pěti různých binárních souborů, které patří ke stejnému typu malwaru. Dá se očekávat, že na místě třech červeně vyznačených bajtů se u dalších souborů stejného typu malwaru mohou vyskytovat jakékoliv hodnoty. Pokud však toto pravidlo použijeme pro detekci tohoto typu malwaru, neuspějeme, neboť pravidlo hledá pouze pět naprosto konkrétních sekvencí.

Řešením zmiňovaných problémů je použít tzv. zástupný symbol (*wildcard*), které se v jazyce YARA zapisuje symbolem *?* a v hexadecimálních řetězcích označuje libovolnou hodnotu jednoho půlbajtu (*nibble*). Jednou možností by bylo pravidla vygenerovaná YaraGenem v případě potřeby manuálně opravit. Toto řešení však samozřejmě není příliš v souladu se snahou vytvořit automatizovaný nástroj, proto byl v rámci této bakalářské práce navržen způsob automatického vyhledávání společných sekvencí.

### Použití zástupných symbolů v podobných sekvencích bajtů na vstupním bodu programu

Z výše uvedeného příkladu se může zdát, že identifikování podobných sekvencí bajtů a použití zástupných symbolů je triviální záležitost. Je třeba si však uvědomit, že praxi se často vyskytnou mnohem komplexnější situace, jako například:

- Mezi soubory na vstupu jich část má na vstupním bodu podobnou sekvenci, ale zbytek má sekvence naprosto odlišné.
- Množina souborů na vstupu má několik podmnožin s podobnými sekvencemi. Napříč podmnožinami však žádná podobnost sekvencí bajtů není.
- Všechny soubory mají na vstupním bodu naprosto odlišné sekvence bajtů.
- „Několik sekvencí je identických, další jsou s nimi téměř identické, další jsou jim velmi podobné a zbylé jsou jim trochu podobné."

Poslední odrážka je záměrně v uvozovkách pro zdůraznění, že takovýto slovní popis je naprosto nedostačující a je potřeba zavést nějaký exaktní algoritmus, který bude pracovat s určitými metrikami a pomocí definovaných prahů (anglicky *threshold*) bude určovat, zda jsou sekvence dostatečně podobné. Správným nastavením prahů je potom možné dosáhnout stavu, kdy výsledek dostatečně generalizuje, ale zároveň není tak obecný, že by byl nepoužitelný.

Výsledný navržený algoritmus iterativně rozděluje vstupní množinu sekvencí (jedna sekvence odpovídá jednomu analyzovanému souboru). Pracuje s pojmem *vzor* (anglicky *pattern*). Vzor vyjadřuje množinu podobných sekvencí a odpovídající sekvenci bajtů obsahující zástupné symboly.

1. Přiřadíme všechny sekvence jednomu vzoru.
2. Opakujeme {
   (a) Pro každý vzor:
      i. Zkusíme najít ve vzoru n-gramy, které dokáží sekvence odpovídající vzoru dobře rozdělit. Pokud se to nepovede, zpracování vzoru končí.
      ii. Rozdělíme vzor na několik nových, podle n-gramu z předchozího bodu.

   } dokud byl nějaký vzor rozdělen.
3. Vyřadíme vzory, které neodpovídají zadaným prahům.

Výběr nejlepších n-gramů pro rozdělení vzoru probíhá následujícím způsobem:

1. Pro každý index od 0 do ( délka sekvence vzoru − délka n-gramu):
   (a) Nalezneme všechny hodnoty n-gramů, které se vyskytují na daném indexu v sekvencích vzoru.
   (b) Vyřadíme ty n-gramy, které nesplňují požadovaná kritéria (*prahy*).

2. Ze všech indexů nalezneme ten, kde n-gramy pokrývají nejvíce sekvencí vzoru (neboli nejvíce vstupních souborů). Pokud n-gramy na daném indexu splňují kritéria (prahy), označíme je za nejlepší n-gramy. Pokud neodpovídají, algoritmus končí neúspěchem.

Výstupem algoritmu hledání podobných sekvencí je množina vzorů. Nakonec je tedy třeba tyto vzory zapsat ve formě YARA pravidla. Bajty vstupního bodu už jsou v tuto chvíli vyřešeny, je ale potřeba doplnit informaci o adrese vstupního bodu.

Opět je snaha o co nejlepší generalizaci. Pokud mají všechny vstupní body daného vzoru stejnou adresu, měla by se tato informace objevit i ve výsledném YARA pravidle. Naopak pokud má každý vstupní bod jinou adresu, nemá smysl adresy vstupního bodu do výsledného YARA pravidla zahrnovat, neboť by toto pravidlo špatně generalizovalo.

Experimentální metodou byla vytvořena hranice, která určuje, zda do výsledného pravidla zahrnovat adresu vstupního bodu.

$$max\ pocet\ adres\ v\ pravidle = \begin{cases} 10 & \text{pokud } p \geq 40 \\ (p/5) + 2 & \text{pokud } p < 40 \end{cases} \quad ;\ p...pocet\ sekvenci\ vzoru$$

Na následujících dvou ukázkách je vidět, že pro vzor s pěti sekvencemi ($p = 5$) budou podle vzorce v pravidle pro tento vzor maximálně tři adresy.

```
/* Pravidlo vygenerovane ze souboru s~temito vstupnimi body:
           bajty                adresa
    00 11 22 33 4E D7 55        0x1111
    00 11 22 33 10 36 55        0x1111
    00 11 22 33 9D FF 55        0x2222
    00 11 22 33 00 7C 55        0x2222
    00 11 22 33 BB 6B 55        0x2222
*/
import "pe"
rule entry_point_example
{
    strings:
       $ep0 = { 00 11 22 33 ?? ?? 55}
    condition:
       (pe.entry_point == 0x1111 and $ep0 at pe.entry_point) or
       (pe.entry_point == 0x2222 and $ep0 at pe.entry_point)
}
```

Kód 4.4: Pravidlo vygenerované novou verzí YaraGenu ze souborů s podobnou sekvencí bajtů na vstupním bodu se dvěma adresami vstupního bodu.

```
/* Pravidlo vygenerovane ze souboru s~temito vstupnimi body:
           bajty              adresa
    00 11 22 33 4E D7 55      0x1111
    00 11 22 33 10 36 55      0x2222
    00 11 22 33 9D FF 55      0x3333
    00 11 22 33 00 7C 55      0x4444
    00 11 22 33 BB 6B 55      0x5555
*/
import "pe"
rule entry_point_example
{
    strings:
       $ep0 = { 00 11 22 33 ?? ?? 55}
    condition:
       // Pravidlo tentokrat neobsahuje adresu vstupniho bodu
       $ep0 at pe.entry_point
}
```

Kód 4.5: Pravidlo vygenerované novou verzí YaraGenu ze souborů s podobnou sekvencí bajtů na vstupním bodu s různými adresami vstupního bodu.

### 4.1.3 Iconhash

Dalším z rysů binárních souborů, které je užitečné analyzovat, je bezpochyby ikona souboru, která se zobrazí uživateli. Pokud mají dva soubory stejnou zobrazovanou ikonu, je to signál, že by celé tyto soubory mohly být podobné a případně by mohly patřit ke stejnému typu malwaru. Bylo tedy navrženo přidání analýzy ikon do nástroje YaraGen, což s sebou přineslo tři úskalí, která budou nyní rozebrána.

Prvním problémem je výběr zobrazované ikony. Jak bylo zmíněno v kapitole 2.1.2, v jednom souboru formátu PE se může nacházet více ikon. Je tedy potřeba vybrat z nich tu, kterou operační systém zobrazí. Algoritmus tohoto výběru byl představen v souběžně zpracovávané bakalářské práci Jakuba Pružince [20].

1. Nalezneme v tabulce zdrojů (*resource table*) první záznam typu `RT_GROUP_ICON`. Tento záznam reprezentuje skupinu ikon, z nichž musíme vybrat zobrazovanou ikonu.

2. Nalezneme nejprioritnější ikonu ze skupiny nalezené v bodě 1. K porovnání priority použijeme tabulku 4.1, přičemž nižší číslo priority znamená vyšší prioritu. Pokud by mělo nejvyšší prioritu více ikon, použijeme tu první (podle pořadí v tabulce zdrojů).

Nyní je vyřešen první problém — je nalezena zobrazovaná ikona. Druhým problémem je, že s daty ikony se nepracuje příliš prakticky, neboť se jedná řádově o několik kilobajtů dat. Proto bylo navrženo použití hashe z dat ikony. Pro hashování je použit algoritmus SHA256 [7] a výsledek je označen slovem *iconhash*.

Třetím problémem přidání *iconhash* do systému YaraGen je fakt, že v době vytváření této práce nepodporoval jazyk YARA použití *iconhash*. YaraGen by tedy mohl generovat YARA pravidla obsahující *iconhash*, tato pravidla by však byla z hlediska nástroje YARA nevalidní. Proto byla v rámci této práce přidána do modulu *pe* funkce *iconhash()*. Její použití vypadá následovně:

```
import "pe"
rule iconhash_example
{
    condition:
       pe.iconhash() == "e2c07eb78684614ef71d6ff75fcb3daa75ea9d85cda
9a4ac9d7d3371724e15ab"
}
```

Kód 4.6: Pravidlo obsahující novou funkci *iconhash()*.

V době dokončování této práce je funkce *pe.iconhash()* používána pouze v lokální verzi nástroje YARA společnosti Avast Software. Je však možné, že v budoucnosti se podaří prosadit zahrnutí této funkce do oficiální veřejně dostupné verze nástroje YARA.

| Priorita | Velikost (v pixelech) | Bitová hloubka |
|---|---|---|
| 0 | 32 | 32 |
| 1 | 24 | 32 |
| 2 | 48 | 32 |
| 3 | 32 | 8 |
| 4 | 16 | 32 |
| 5 | 64 | 32 |
| 6 | 24 | 8 |
| 7 | 48 | 8 |
| 8 | 16 | 8 |
| 9 | 64 | 8 |
| 10 | 96 | 32 |
| 11 | 96 | 8 |
| 12 | 128 | 32 |
| 13 | 128 | 8 |
| 14 | 256 | 32 |
| 15 | 256 | 8 |
| 16 | Ostatní | |

Tabulka 4.1: Tabulka priorit ikon. Pozor: velikost ikony udává zároveň výšku a šířku a předpokládá tedy, že ikona je čtvercová. Pokud čtvercová není, automaticky má nejnižší prioritu.

### 4.1.4 Další přidané či upravené analýzy

Kromě výše zmíněných přidaných analýz došlo také k úpravě či doplnění stávajících analýz, uvedených v sekci 3.3.2. Zde je telegrafický přehled všech změn, úprav a rozšíření statických analýz.

#### Omezení délky řetězců

Ing. Marek Milkovič ve své diplomové práci [18] představil metodu, jak v binárních souborech detekovat řetězce. Ve většině případů tato metoda funguje velice dobře a dokáže určit, které sekvence bajtů jsou řetězce a které nikoliv. V praxi ovšem nastaly případy, kdy byla

za řetězec prohlášena velmi dlouhá sekvence (i stovky tisíc bajtů). Taková sekvence nejen že není skutečným řetězcem, ale při použití YARA pravidla obsahujícího tak dlouhou sekvenci navíc došlo k přetečení bufferu. Na základě toho byl do analýzy řetězců přidán konfigurovatelný parametr *max_length*, kterým je možné nastavit maximální délku řetězců. Řetězce delší než *max_length* jsou zkráceny na tuto délku. Výchozí hodnota parametru *max_length* je nastavena na 80.

### Whitelist běžných jmen sekcí

Analýza jmen sekcí ve staré verzi YaraGenu často nepřinášela příliš hodnotné výsledky, neboť výsledné YARA pravidlo obsahovalo jména sekcí natolik běžná, že pravidlo bylo příliš obecné. Jména jako *.text* či *.data* jsou sice používána pro názvy sekcí malwaru, ale také pro názvy sekcí jakýchkoliv jiných neškodných binárních souborů formátu PE. Proto byl vytvořen whitelist běžných jmen sekcí. Pokud je v souboru nalezena sekce, jejíž jméno se zároveň objevuje ve whitelistu, je tato sekce ignorována a její jméno není generováno do výsledného pravidla jazyka YARA.

### Otisk podepisujícího

Do nástroje YARA byla ve verzi 3.8 přidána možnost detekce na základě otisku podepisujícího (*thumbprint of signature*). Jedná se o SHA1 [8] hash vypočtený z certifikátu. Analýza otisku podepisujícího byla tedy přidána i do YaraGenu.

```
import "pe"
rule thumbprint_example
{
 condition:
   pe.signatures[0].thumbprint == "f7c3bc1d808e04732adf679965ccc34ca7ae3441"
}
```

Kód 4.7: Pravidlo obsahující otisk podepisujícího.

### TypeLib identifikátor

TypeLib identifikátor je globálně unikátní identifikátor (GUID), který se vyskytuje v souborech formátu PE pro platformu .NET. Původní verze YaraGenu byla vytvářena v dobách, kdy YARA ještě neměla modul *dotnet* a nepodporovala tak parsování souborů pro tuto platformu. Vše se však změnilo s verzí YARA 3.6. Proto bylo v YaraGenu upraveno generování TypeLib identifikátoru tak, aby využívalo nový atribut *dotnet.typelib*, jak je vidět na ukázce 4.8.

### Entropie sekcí a *overlay*

YARA ve verzi 3.3 přinesla možnost používat v podmínkách výpočet entropie [19]. Vzhledem k tomu, že Jakub Pružinec v rámci své bakalářské práce [20] přidal výpočet entropie do nástroje Fileinfo, je logické přidat podporu pro analýzu entropie i do YaraGenu. Konkrétně se jedná o analýzu entropie sekcí a tzv. *overlay*. *Overlay* je část dat, která je připojena na konec spustitelného souboru a není nijak odkazována z hlaviček souboru.

Pro účely analýzy malware je podstatné zjištění, že některá sekce či *overlay* nabývají vysoké entropie. To totiž obvykle znamená, že daná část souboru byla zkomprimována či zašifrována.

Entropie obvykle nabývá hodnot v intervalu $< 0; 1 >$, avšak YARA využívá variantu, kdy se tento interval lineárně mapuje na interval $< 0; 8 >$. Experimentálně bylo určeno, že do výsledného YARA pravidla bude entropie generována, pokud přesáhne hodnotu 6,8. Tato hodnota může být konfigurována pomocí parametru *min_entropy*. Příklad pravidla obsahujícího entropii je v kódu 4.9.

```
// Stara verze
rule typelib_example_old
{
    strings:
        $type_lib_id_g0_p100 = "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    condition:
        $type_lib_id_g0_p100
}

// Nova verze
import "dotnet"
rule typelib_example_new
{
    condition:
        dotnet.typelib == "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb"
}
```

Kód 4.8: Porovnání staré a nové verze pravidla obsahujícího TypeLib identifikátor.

```
import "pe"
import "math"
rule entropy_example
{
    condition:
        math.entropy(pe.sections[0].raw_data_offset,
                     pe.sections[0].raw_data_size) > 6.8
        or
        math.entropy(pe.overlay.offset, pe.overlay.size) > 6.8
}
```

Kód 4.9: Pravidla obsahující entropii.

## 4.2 Generování behaviorálních rysů

Podstatná část přínosu této práce spočívá v přidání analýz behoviorálních rysů do nástroje YaraGen. Tyto rysy jsou při detekci malwaru považovány za lepší a spolehlivější.

Jako vstup používá YaraGen pro behaviorální analýzy reporty ze sandboxu Cuckoo ve formátu JSON. Nástroj YARA obsahuje modul *cuckoo*, který je kompatibilní s těmito reporty. V současné verzi 3.10 však modul *cuckoo* obsahuje poměrně malé množství funkcí a jeho použití je tak omezené. Společnost Avast Software však používá vlastní verzi nástroje

YARA, která v modulu *cuckoo* nabízí širší paletu použitelných funkcí. Všechny tyto funkce mají jeden parametr, kterým je regulární výraz.

### 4.2.1 Analýzy rysů získaných z Cuckoo

V této podkapitole budou představeny všechny behaviorální rysy, jejichž analýza byla v rámci této práce přidána do nástroje YaraGen. Tento seznam se téměř shoduje se seznamem funkcí, které v Avast Software podporuje modul *cuckoo*.

#### Pojmenované objekty

Pojmenované objekty poskytují procesům jednoduchý způsob, jak sdílet tzv. *object handles* [14]. Jeden proces může vytvořit pojmenovaný objekt, k němuž poté další procesy mohou přistupovat pomocí jeho jména. Při analyzování pojmenovaných objektů je předmětem analýzy právě jejich jméno. Pojmenované objekty se používají především pro synchronizaci procesů. Do této kategorie se dá zařadit mutex, semafor, event, atom, sekce, úloha (*job*) a časovač (*timer*).

Druhou kategorii tvoří pojmenované roury (*pipe*) a tzv. *mailsloty*. Oba tyto typy pojmenovaných objektů se používají pro meziprocesovou komunikaci.

Třetí kategorií pojmenovaných objektů, které jsou sandboxem Cuckoo (a následně i nástrojem YaraGen) analyzovány, jsou služby (*service*). Konkrétně se jedná o vytvořené a spuštěné služby.

```
import "cuckoo"
rule named_objects_example
{
    condition:
        cuckoo.sync.mutex(/mutex_name/) or
        cuckoo.sync.semaphore(/semaphore_name/) or
        cuckoo.sync.event(/event_name/) or
        cuckoo.sync.atom(/atom_name/) or
        cuckoo.sync.section(/section_name/) or
        cuckoo.sync.job(/job_name/) or
        cuckoo.sync.timer(/timer_name/) or
        cuckoo.filesystem.pipe(/pipe_name/) or
        cuckoo.filesystem.mailslot(/mailslot_name/) or
        cuckoo.process.created_service(/service_name/) or
        cuckoo.process.started_service(/service_name/)
}
```

Kód 4.10: Pravidla obsahující ukázku všech pojmenovaných objektů, které YaraGen podporuje.

#### Přístup k souborovému systému

Jedná se o čtení, zápis či mazání souboru. Kromě konkrétních funkcí pro jednotlivé typy přístupu nabízí YARA v Avast Software ještě obecnější funkci *file_access()*, která vyjadřuje vykonání jakékoliv z těchto tří operací. Tato obecnější funkce je použita, pokud program

stejný soubor četl, zapisoval do něj a smazal ho. Funkce *file_access()* je také použita v případě, kdy v reportu ze sandboxu Cuckoo není přesně specifikováno, která ze tří operací byla s daným souborem prováděna.

```
import "cuckoo"
rule file_access_example
{
    condition:
        cuckoo.filesystem.file_read(/file1/) or
        cuckoo.filesystem.file_write(/file2/) or
        cuckoo.filesystem.file_delete(/file3/) or
        cuckoo.filesystem.file_access(/file4/)
}
```

Kód 4.11: Pravidla obsahující přístup k souborovému systému.

### Přístup k registrovým klíčům

Přístup k registrovým klíčům je řešen obdobně, jako přístup k souborovému systému.

```
import "cuckoo"
rule key_access_example
{
    condition:
        cuckoo.registry.key_read(/key1/) or
        cuckoo.registry.key_write(/key2/) or
        cuckoo.registry.key_delete(/key3/) or
        cuckoo.registry.key_access(/key4/)
}
```

Kód 4.12: Pravidla obsahující přístup k registrovým klíčům.

### Vykonané příkazy

Program může během svého běhu vykonávat externí příkazy, jako například spouštění jiných programů. Cuckoo umí tyto vykonané příkazy zaznamenat, pro detekci se pak v jazyce YARA použije funkce *executed_command()*.

```
import "cuckoo"
rule executed_command_example
{
    condition:
        cuckoo.process.executed_command(/command_name/) or
}
```

Kód 4.13: Pravidlo obsahující vykonané příkazy.

### Síťová komunikace

Cuckoo umožňuje analyzovat také síťovou komunikaci daného programu. YaraGen se zaměřuje konkrétně na HTTP požadavky a vyhledávání v systému DNS. Jsou použity funkce

z podmodulu *cuckoo.network*. Pokud program vykonal HTTP požadavek metodou GET či POST, je použita funkce *http_get()* nebo *http_post()*. Pokud program vykonal HTTP požadavek jinou metodou (např. HEAD, PUT, atd.), je použita obecnější funkce *http_request()*, neboť konkrétní funkce podle metody požadavku jsou k dispozici pouze pro GET a POST. Obecná funkce *http_request()* je použita také v případě, že program poslal na stejnou URI jak GET požadavek, tak také POST požadavek.

```
import "cuckoo"
rule network_example
{
    condition:
        cuckoo.network.http_get(/uri1/) or
        cuckoo.network.http_post(/uri2/) or
        cuckoo.network.http_request(/uri3/) or
        cuckoo.network.dns_lookup(/uri4/)
}
```

Kód 4.14: Pravidlo obsahující síťovou komunikaci.

#### *Resolved API*

Při načítání funkcí z dynamicky načítaných knihoven (*dll*) se někdy používá speciální způsob překladu jmen těchto funkcí [3]. Pomocí funkce *resolved_api()* je možné v jazyku YARA analyzovat, ke kterým funkcím (API) program tímto způsobem přistupoval. Jednotlivá API se zapisují jako název knihovny, znak '!' a název funkce.

```
import "cuckoo"
rule resolved_api_example
{
    condition:
        cuckoo.process.resolved_api(/kernel32\.dll!GetTickCount/) or
}
```

Kód 4.15: Pravidlo obsahující získaná API.

#### Signatury

Cuckoo umí popsat chování programu i s vyšší mírou abstrakce. K tomu se používají tzv. signatury. Každá signatura má jméno a popis. Program odpovídá určité signatuře, pokud při běhu v sandboxu Cuckoo splní podmínky této signatury. Příkladem jmen signatur může být například *ransomware_file_modifications* či *copies_self*.

```
import "cuckoo"
rule signature_example
{
    condition:
        cuckoo.signature.name(/infostealer_keylog/) or
}
```

Kód 4.16: Pravidlo obsahující signaturu.

### 4.2.2 Generování YARA pravidel obsahujících behaviorální rysy

Ve společnosti Avast Software jsou pro zápis pravidel formátu YARA stanoveny určité konvence. Jedna z nich určuje rozdělení pravidel podle jejich spolehlivosti. Spolehlivost se určuje podle rysů, které YARA pravidlo obsahuje. Existují celkem čtyři kategorie:

1. `known_named_objects` — Pojmenované objekty
2. `known_behavior_high` — Ostatní behaviorální rysy
3. `known_behavior_low` — Méně důvěryhodná behaviorální pravidla obsahující stejné rysy, jako `known_behavior_high`
4. `known_sequences` — Statické rysy

```
import "cuckoo"
import "pe"
rule example_known_named_objects
{
    condition:
        cuckoo.sync.mutex(/mutex1/)
}

rule example_known_behavior_high
{
    condition:
        not example_known_named_objects and (
            cuckoo.network.http_request(/something.evil.com/) or
            cuckoo.filesystem.file_read(/evil_file.txt/)
        )
}

rule example_known_sequences
{
    condition:
        not example_known_named_objects and
        not example_known_behavior_high and (
            pe.imphash() == "b8bb385806b89680e13fc0cf24f4431e" or
            pe.number_of_sections == 13
        )
}
```

Kód 4.17: Soubor formátu YARA obsahující tři YARA pravidla.

Rozlišit automatickými metodami, kdy je behaviorální pravidlo spolehlivější a kdy méně, je značně problematické. Proto bylo navrženo v nástroji YaraGen nepoužívat kategorii `known_behavior_low` a generovat pravidla pouze zbylých třech kategorií.

Další konvence říká, že pokud jeden soubor obsahuje více YARA pravidel, musí být tato pravidla řetězena určitým speciálním způsobem.

- Pravidla musí být seřazena podle spolehlivosti.

- Analyzovaný vstupní soubor musí vždy odpovídat maximálně jednomu YARA pravidlu. Pokud například binární soubor odpovídá pravidlu typu `known_named_objects`, nesmí už tento binární soubor odpovídat žádnému dalšímu pravidlu zapsanému v tom samém souboru s YARA pravidly.

Druhé z předchozích podmínek se dosahuje tak, že do pravidel je přidána negace všech předchozích pravidel. Jakmile tak vstupní soubor odpovídá nějakému pravidlu, už nemůže odpovídat následujícím pravidlům.

Generátor behaviorálních rysů pro nástroj YaraGen byl navržen s respektem k předchozím pravidlům. Na ukázce 4.17 je možné vidět, jak vypadá vygenerovaný soubor formátu YARA obsahující tři pravidla.

### 4.2.3 Generování regulárních výrazů z behaviorálních rysů

Jak již bylo zmíněno, všechny funkce jazyka YARA z modulu *cuckoo*, které jsou generované nástrojem YaraGen, mají jeden parametr, kterým je regulární výraz. Ve všech výše zmíněných příkladech byl však použit takový regulární výraz, který odpovídá pouze jednomu řetězci. V první fázi bylo generování behaviorálních rysů opravdu takto implementováno — jeden název behaviorálního rysu odpovídal jednomu regulárnímu výrazu ve vygenerovaném YARA pravidle. Název behaviorálního rysu byl ještě obalen znaky pro začátek(^) a konec($) řetězce. Tím je zaručeno, že daný regulární výraz odpovídá opravdu pouze jednomu požadovanému řetězci.

Byla by však škoda se omezit na takovéto jednoduché regulární výrazy. Vznikl proto požadavek na generování složitějších regulárních výrazů z behaviorálních rysů. Na příkladu je možné vidět značné zpřehlednění pravidla při použití regulárního výrazu vytvořeného z několika názvů mutexů.

```
import "cuckoo"
rule regex_example
{
    condition:
        // Mapovani 1:1 mezi retezci a regularnimi vyrazy
        cuckoo.sync.mutex(/^mutexA$/) or
        cuckoo.sync.mutex(/^mutexB$/) or
        cuckoo.sync.mutex(/^mutexC$/) or

        // Pouziti slozitejsiho regularniho vyrazu
        cuckoo.sync.mutex(/^mutex[ABC]$/)
}
```

Kód 4.18: Pravidlo obsahující regulární výrazy.

Proto byl navržen algoritmus generování regulárních výrazů z množiny řetězců. Hned na úvod je třeba podotknout, že se jedná o velice náročný úkol výzkumného charakteru. Cílem totiž nebylo vytvořit pouze jeden obrovský regulární výraz, který by pokrýval všechny původní řetězce. Vzhledem k tomu, že výstup z YaraGenu prochází ještě manuální úpravou analytiků, důležitým požadavkem bylo, aby byl regulární výraz čitelný pro člověka. Tento cíl se nakonec podařilo splnit pouze částečně, v některých případech je výsledný regulární výraz zbytečně složitý a komplikovaný. Proto je i do budoucna v plánu na tomto úkolu dále pracovat a generátor regulárních výrazů vylepšovat.

Algoritmus generování regulárních výrazů je následující:

1. Všechny řetězce uložíme do prefixového stromu (trie) [2]

2. Vytvoříme z trie deterministický acyklický konečný stavový automat (*deterministic acyclic finite state automaton, DAFSA*) neboli orientovaný acyklický slovní graf (*directed acyclic word graph*, *DAWG*), který má jeden finální stav, ze kterého už nevedou žádné další přechody.

3. Tento orientovaný acyklický slovní graf projdeme od koncového uzlu směrem po počáteční. Pro každý uzel vytvoříme regulární výraz. Regulární výraz pro koncový uzel je prázdný řetězec. Pro každý další uzel získáme regulární výraz za pomoci regulárních výrazů následujících uzelů.

Bližší pozornost si zaslouží druhý bod algoritmu, převod trie na orientovaný acyklický slovní graf. Byl použit algoritmus, který vychází z metody pro minimalizaci deterministického acyklického stavového automatu v lineárním čase, kterou navrhl Dominique Revuz [21].

Ještě před samotným popisem algoritmu je nutné definovat pojem *výška uzlu*. Pro uzel $u$ je výška uzlu $u$ rovna délce nejdelší cesty začínající v $u$ a končící v koncovém uzlu.

1. Pro $i$ jdoucí od 0 do výšky počátečního uzlu:

   (a) Vezmeme všechny uzly s výškou $i$ a najdeme mezi nimi duplicitní uzly. Duplicitní uzly jsou takové, které mají identické přechody do dalších uzlů.

   (b) Každou skupinu duplicitních uzlů spojíme do jednoho — vybereme si z této skupiny jeden uzel, který ponecháme, a ostatní odstraníme. Přechody, které směřovaly do odstraněných uzlů, upravíme tak, aby směřovaly do ponechaného uzlu.

Tento algoritmus se od Revuzova algoritmu liší pouze v řazení uzlů, které je použito kvůli nalezení skupin duplicitních uzlů. Revuzův algoritmus používá přihrádkové řazení (*bucket sort*), zatímco v této práci je použito řazení pomocí funkce ze standardní knihovny jazyka C++.

# Kapitola 5

# Implementace generování detekčního vzoru

Tato kapitola se zabývá implementací metod navržených v předchozí kapitole. Tyto metody byly integrovány do nástroje YaraGen.

## 5.1 Architektura nástroje YaraGen

Před samotným popisem implementace nových metod je třeba stručně představit důležité prvky implementace nástroje YaraGen. Podrobnější informace lze nalézt v diplomové práci Ing. Milkoviče [18]. (V některých drobnostech se informace uvedené v jeho diplomové práci liší od současného stavu věcí, avšak hlavní prvky návrhu zůstaly zachovány.)

Nástroj YaraGen je implementován v jazyce C++ podle revize C++17 [9]. Pro pohodlnou kompilaci je použit systém CMake [12].

Nástroj je implementován modulárně a umožňuje snadnou rozšiřitelnost. Funkcionalita je implementována v knihovně *yaragenlib*. Program *yaragen* je konzolová aplikace, která představuje rozhraní této knihovny.

Knihovna *yaragenlib* se dá rozdělit na čtyři části:

- Extraktor — jeho úkolem je vyextrahovat informace ze vstupních souborů. Rozhraní extraktoru určuje abstraktní třída `FileLoader` a její abstraktní metoda `load(FileInformation&)`. Od této třídy dědí třída `FileFormatLoader`, která implementuje načítání dat ze vstupních souborů ve formátu JSON. Informace ukládá do instance třídy `FileInformation`. Tato třída uchovává všechny informace o jednom souboru.

- Analyzátor — stará se o provedení analýz a určení, které hodnoty se dostanou do výsledného vygenerovaného detekčního vzoru. Základem je třída `AnalysisEngine`, která nejprve pro každý vstupní soubor spustí analýzy pomocí `runAnalysis` a poté analýzy dokončí funkcí `finalizeAnalysis`.
  Analýza je implementována v abstraktní třídě `Analysis`. Analýza každého rysu má svou třídu poděděnou z třídy `Analysis`. Výsledky analýzy jsou uloženy v instancích tříd, které dědí z abstraktní třídy `AnalysisResult`. Nastavení analýz jsou uchována v třídách, které dědí z abstraktní třídy `AnalysisSettings`
  `CoverageResolver` je třída, která řeší pokrytí vstupních souborů analyzovanými hodnotami.

- Generátor — jeho úkolem je vygenerovat YARA pravidla na základě výstupu analyzátoru. Základem je abstraktní třída `Generator`. Její podtřídy jsou konkrétní generátory. V původní verzi YaraGenu byl pouze jeden generátor, `StaticRuleGenerator`. Pro generování behaviorálních informací byly přidány dva další generátory (viz 5.2).
- Konfigurace — třída `Config` uchovávající veškerá nastavení a konfigurace nástroje.

## 5.2 Implementace nových analýz

Přidání statických analýz probíhalo tak, že byly vytvořeny odpovídající třídy pro analýzu (dědící z `Analysis`), pro výsledek analýzy (dědící z `AnalysisResult`) a pro nastavení analýzy (dědící z `AnalysisSettings`).

Více úprav bylo potřeba udělat při přidání behaviorálních analýz. Byly přidány dva nové generátory — `NamedObjectsRuleGenerator` a `BehaviorHighRuleGenerator`. Je pro ně použit návrhový vzor *Visitor*. Každý z těchto generátorů implementuje virtuální metody `visit(ConcreteAnalysisResult*)`, kde argumentem je ukazatel na výsledek konkrétní analýzy (např. `MutexAnalysisResult*`). Tyto metody implementují generování výsledků jednotlivých analýz. Používají k tomu knihovnu *yaramod*. V každé metodě `visit` je z výsledku analýzy vytvořen výraz, který je přidán do pravidla pomocí metody `addConditionTerm`

Dále byla upravena metoda `generateRule` v abstraktní třídě `Generator`. V ní byla implementována logika, která vytváří řetězec YARA pravidel typu `known_named_objects`, `known_behavior_high` a `known_sequences`, jak bylo popsáno v 4.2.2.

## 5.3 Generování zástupných symbolů do podobných sekvencí bajtů

Generování zástupných symbolů (*wildcard*) pro analýzu vstupního bodu programu bylo implementováno ve třídě `CommonSequenceFinder`. Té jsou v konstruktoru předány vstupní sekvence a potřebné údaje o konfiguraci. Veškeré hledání podobných sekvencí a vkládání zástupných symbolů se odehrává v metodě `findCommonSequences()`.

Dále jsou použity čtyři pomocné třídy:

- `Sequence` reprezentuje jeden n-gram, konkrétně 5 bajtů na určitém indexu.
- `SequenceTable` reprezentuje tabulku všech n-gramů na určitém indexu. Obsahuje atribut `_sequences` typu `std::set<Sequence>`.
- `PositionSequenceTable` reprezentuje tabulku všech n-gramů na všech indexech pro daný vzor (`Pattern`). Obsahuje atribut `_sequenceTables` typu `std::vector<SequenceTable>`.
- `Pattern` reprezentuje vzor, neboli množinu vstupních sekvencí, které jsou si podobné.

Hledání podobných sekvencí probíhá tak, že nejprve jsou všechny sekvence přiřazeny jednomu vzoru. Pomocí metody `findBestSequenceTable` je nalezena tabulka n-gramů, podle které je možné vzor dobře rozdělit. Rozdělení vzoru na několik menších se provede opakovaným voláním metody `splitBySequence(Sequence)`, kde argumentem jsou postupně všechny n-gramy z tabulky n-gramů.

Metoda `splitBySequence` má jeden parametr, kterým je n-gram. Metoda funguje tak, že rozdělí jeden vzor (`Pattern`) na dva. Jeden nový vzor bude obsahovat sekvence, ve kterých se na daném indexu vyskytuje zadaný n-gram. Druhý vzor bude obsahovat sekvence, ve kterých se na daném indexu vyskytuje jakýkoliv jiný n-gram.

Celý proces rozdělování vzorů se opakuje, dokud metoda `findBestSequenceTable` nalézá nějakou tabulku n-gramů, podle které se nějaký vzor dá rozdělit.

## 5.4 Generování regulárních výrazů

Hlavní třídou zastřešující rozhraní generátoru regulárních výrazů je třída `RegexGenerator`. Její metoda `addWord` přidá do generátoru jeden řetězec. Když jsou všechny řetězce přidány, pomocí metody `toRegexes` jsou vygenerovány regulární výrazy.

Instance třídy `RegexGenerator` v sobě uchovávají orientovaný acyklický slovní graf (*DAWG*). Jeden uzel tohoto grafu je reprezentován objektem třídy `DawgNode`. Celý graf je pak reprezentován datovým typem `std::vector<DawgNode>`. Provázání jednotlivých uzlů tedy není realizováno ukazateli, ale pomocí indexů do tohoto vektoru.

V metodě `toRegexes` je nejprve zavolána metoda `minimize`, která minimalizuje graf *DAWG*. Poté je tento graf převeden na regulární výraz. K tomu je použita abstraktní třída `Expression`, která reprezentuje právě regulární výraz. Z ní dědí pět tříd:

- `Literal` — řetězcový literál
- `Alternation` — výběr mezi více možnostmi; např. *(ABC|XYZ)*
- `ZeroOrOneRepetition` — nula nebo jedno opakování; např *(ABC)?*
- `Concatenation` — konkatenace dvou výrazů; např. *A(BC|D)* je konkatenace literálu a alternace
- `CharClass` — jeden ze skupiny znaků; např. *[ABC]*

Některé třídy (`Alternation`, `ZeroOrOneRepetition` a `Concatenation`) obsahují ve svém atributu `_value` ukazatel na další výraz či výrazy typu `Expression`. Vzniká tak celý strom, který reprezentuje složitější regulární výraz.

Další pomocnou třídou je třída `RegexString`, která představuje regulární výraz. Na rozdíl od `Expression` je tento regulární výraz reprezentován ve formě řetězce, nikoliv v abstraktní formě stromu. Objekty třídy `RegexString` kromě samotného řetězce obsahují ještě dva další atributy — `_numOfNestedAlternations` a `_numOfNestedZeroOreOneRepetitions`. Tyto dva atributy uchovávají počet vnořených alternací a vnořených nepovinných výskytů (metaznak *?*). Tyto dva atributy mají za cíl splnění jednoho z požadavků na generátor regulárních výrazů, tedy aby byly výrazy čitelné pro člověka. Čitelnost je stanovována pomocí porovnání s maximálními povolenými hodnotami počtu vnořených alternací a nepovinných výskytů.

Převádění regulárního výrazu z abstraktní formy (`Expression`) na textovou reprezentaci (`RegexString`) se děje ve virtuální metodě `toStrings`. Tato metoda může vracet více než jeden výraz. Je to proto, že kvůli čitelnosti je vhodnější mít dva menší regulární výrazy než jeden velký. Obsah metody se liší podle konkrétní třídy. Například v metodě `Literal::toStrings` dochází pouze k *vyescapování* obsaženého literálu. Naopak v `Alternation::toStrings` jsou rekurzivně zavolány metody `toStrings` jednotlivých podvýrazů alternace. Poté jsou nalezeny společné sufixy pomocí třídy `CommonSuffixResolver`.

Nakonec je vytvořena alternace z jednotlivých výrazů, které jsou spojené na základě společných sufixů.

## 5.5 Spouštění nástroje YaraGen a skript yaragen.py

Nástroj YaraGen je konzolová aplikace. Spouští se příkazem

*yaragen* [*Přepínače*] [*Vstupní soubory*]

Vstupními soubory jsou reporty z nástroje Fileinfo nebo ze sandboxu Cuckoo ve formátu JSON. Vstupním souborem může být případně i složka, která je poté procházena rekurzivně a jsou analyzovány všechny soubory, které obsahuje.

Pokud je potřeba zanalyzovat binární soubor, je potřeba nad ním tedy manuálně spustit analýzy nástroji Fileinfo a Cuckoo. To je samozřejmě poněkud nepraktické. Proto byl vytvořen skript `yaragen.py`. Je napsán v jazyce Python ve verzi 3 a jeho úkolem je usnadnit analýzu binárních souborů pomocí nástroje YaraGen.

Skript se spouští příkazem

*./yaragen.py* [*Přepínače skriptu*] [*Vstupní soubory*] [-- *Přepínače YaraGenu*]

Velkou změnou oproti spouštění nástroje YaraGen je to, že vstupními soubory jsou pro skript `yaragen.py` přímo binární soubory, které je potřeba analyzovat. O vše ostatní se postará sám skript.

1. Pro každý vstupní soubor: Pokud je ve složce, ve které se nachází vstupní binární soubor, uložen soubor stejného jména s příponou *.json*, použije se tento JSON soubor. V opačném případě je nad vstupním souborem spuštěn nástroj Fileinfo a je uložen JSON report.

2. Pro každý vstupní soubor: Proběhne stejné vyhledávání, jako v bodě 1., s tím rozdílem, že je tentokrát vyhledán soubor s příponou *.cuckoo.json*. Pokud je nalezen, je použit jako vstup YaraGenu. V opačném případě je report z Cuckoo stažen ze serveru pomocí objektu třídy `CuckooDownloader`. Vzhledem k tomu, jak jsou výsledky analýz z Cuckoo ve společnosti Avast Software uloženy, je výsledek analýzy hledán v tomto pořadí (pokud je vyhledávání v některém bodě úspěšné, hledání tím končí):

   (a) Samba úložiště analýz, ve kterém se nacházejí výsledky nedávných analýz.

   (b) xBR databáze, ve které se nacházejí některé údaje z analýz. Při použití xBR databáze je JSON report zrekonstruován, i když v některých případech se nepodaří původní JSON report zrekonstruovat úplně.

   (c) Pokud je zapnut přepínač `submit`, je binární soubor odeslán na server do sandboxu Cuckoo na novou analýzu. Jinak končí hledání JSON reportu z Cuckoo neúspěchem.

3. Je spuštěn nástroj YaraGen, přičemž vstupem mu jsou všechny získané JSON reporty s příponami *.json* a *.cuckoo.json*

4. Jsou smazány všechny získané JSON reporty (pokud není zadán přepínač `keep-jsons`).

Byla také implementována třída `Logger`, která se stará o informování uživatele o běhu skriptu. Pro přehledné zobrazení ve formě indikátoru průběhu (*progress bar*) používá modul `tqdm` [6].

Pro správné fungování skriptu `yaragen.py` je třeba v souboru *config.ini* vyplnit cestu k nástroji Fileinfo a k nástroji YaraGen a dále vyplnit přístupové údaje, které používá xBR databáze.

V následující tabulce je přehled přepínačů skriptu `yaragen.py`:

| **Přepínač** | **Význam** |
|---|---|
| `-h\|--help` | Vypíše nápovědu |
| `-t\|--threads` N | Počet použitých jader procesoru |
| `-k\|--keep-jsons` | Po skončení nesmaže JSON reporty |
| `-g\|--only-generate-jsons` | Pouze získá JSON reporty, ale nespouští YaraGen |
| `--submit` | Soubory, pro které nebyl nalezen výsledek analýzy v Cuckoo, odešle do Cuckoo na analýzu |
| `--force-submit` | Odešle na analýzu do Cuckoo i ty soubory, pro které by výsledek analýzy byl nalezen |

Tabulka 5.1: Přepínače skriptu `yaragen.py`

## 5.6 Testování

YaraGen je testován sadou testů, které jsou stejně jako samotný nástroj implementovány v jazyce C++. Jedná se o jednotkové a integrační testy. Jednotkové testy se zaměřují na testování jednotlivých komponent systému, integrační testují celkovou funkčnost knihovny `yaragenlib`.

Tak jak byla v průběhu práce přidávána funkcionalita nástroje YaraGen, byly postupně přidávány také testy. K původním 22 jednotkovým testům bylo přidáno 48 dalších, nyní jich je tedy 70. V případě integračních testů byl počet zvýšen o 55 z původních 44 na současných 99.

Vzhledem k tomu, že nástroj je již používán v praxi, bylo provedeno také několik experimentů nad reálnými vzorky. Tyto experimenty sice nemohou sloužit jako exaktní ukazatel kvality nástroje, ale mohou naznačit, jak spolehlivý a použitelný je. Při všech experimentech byl nástroj YaraGen spouštěn nad shluky (*clustery*) z interního systému společnosti Avast Software zvaného Clusty. Tento systém shlukuje soubory na základě společných vlastností.

V prvním experimentu byl YaraGen spuštěn nad soubory, které odpovídaly nějakému Yara pravidlu používanému v rámci společnosti Avast Software pro detekci malwaru. Smyslem experimentu bylo ověřit, nakolik se pravidlo vygenerované YaraGenem podobá původnímu pravidlu vytvořenému manuálně a používanému v praxi. Výsledek experimentu byl takový, že vygenerované pravidlo vždy obsahovalo hodnoty, které byly v původním pravidlu v konjunkci. U hodnot, které byly v původním pravidle v disjunkci, byla úspěšnost menší. Je to způsobeno tím, že hodnoty, které analytik zapíše do disjunkce, pokrývají často jen malou část souborů. YaraGen, pro který je zásadní právě to, kolik procent souborů hodnota pokrývá, tak tyto hodnoty s malým procentuálním pokrytím někdy ignoruje.

Ve druhém experimentu bylo ověřeno, jak často se podaří vygenerovat jednotlivé typy pravidel podle podkapitoly 4.2.2. YaraGen byl spuštěn nad deseti náhodně vybranými

shluky. Jak je vidět v tabulce 5.2, vždy se podařilo vygenerovat statické pravidlo. U behaviorálních byla úspěšnost menší. Důvod je prostý — soubory v některých použitých shlucích neměly žádné behaviorální rysy, které YaraGen analyzuje, takže pravidlo nebylo z čeho vygenerovat.

| Typ pravidla | Počet pravidel | Procentuální úspěšnost |
|---|---|---|
| `known_named_objects` | 6 | 60 % |
| `known_behavior_high` | 7 | 70 % |
| `known_sequences` | 10 | 100 % |

Tabulka 5.2: Počet pravidel vygenerovaných z deseti shluků podle typu.

V posledním testu byla ověřena generalizace vygenerovaných pravidel. Opět bylo použito 10 náhodných shluků ze systému Clusty, které dohromady obsahovaly téměř tisíc binárních souborů. Tentokrát ale nebyl YaraGen spuštěn nad celými shluky, ale vždy pouze nad 30 % souborů ve shluku. Poté byly pomocí vygenerovaného pravidla zanalyzovány všechny soubory ve shluku pomocí nástroje YARA. Správná generalizace by měla vypadat tak, že pravidlo bude pokrývat všechny soubory shluku, z jehož části bylo vygenerováno. V tabulce 5.3 je uvedeno porovnání výsledků staré a současné verze YaraGenu. Je vidět, že současná verze předčí verzi původní. Nejde jen o vyšší celkové pokrytí souborů shluku. Důležité je i zjištění, že většina souborů je detekována pravidlem typu `known_named_objects`, které je obecně považováno za nejspolehlivější a obsahuje rysy, které se těžko zatemňují (obfuskují).

| | Stará verze | Současná verze |
|---|---|---|
| `known_named_objects` | 0 % | 84 % |
| `known_behavior_high` | 0 % | 8 % |
| `known_sequences` | 93 % | 6 % |
| Celkem | 93 % | 98 % |

Tabulka 5.3: Porovnání staré a nové verze YaraGenu pro třetí experiment. Hodnoty vyjadřují procentuální pokrytí shluku pravidlem v dané kategorii. Pravidla byla vygenerována ze 30 % shluku.

# Kapitola 6

# Závěr

Cílem této práce bylo vytvořit nové metody generování vzorů pro detekci škodlivého kódu a vylepšit metody stávající. Tento cíl byl naplněn. Bylo navrženo generování nových rysů binárních souborů do detekčních pravidel. Také byly navrženy úpravy ve stávajícím generování rysů. Tyto úpravy vedou ke generování kvalitnějších detekčních vzorů.

Všechny představené metody byly implementovány do systému YaraGen, který je vyvíjen a používán společností Avast Software a jehož úkolem je generování detekčního vzoru v jazyce YARA z množiny binárních souborů.

Do systému YaraGen byly přidány analýzy čtyř statických rysů. Analýzy dalších pěti statických rysů byly upraveny a vylepšeny. Za zmínku stojí například navržený algoritmus na hledání podobných sekvencí, který je použit v analýze bajtů na vstupním bodu programu.

Dále byly přidány analýzy pro 18 behaviorálních rysů, včetně pojmenovaných objektů, přístupu k souborovému systému či síťové komunikace. Jedná se o rysy, které se v praxi běžně pro detekci škodlivého kódu používají a jsou považovány za nejspolehlivější. Celkem je pro tyto behaviorální rysy generováno 26 funkcí jazyka YARA z modulu *cuckoo*. Pro účely generování behaviorálních rysů byl představen algoritmus na vytváření regulárních výrazů z množiny řetězců. Tento algoritmus by se dal označit za nejvíce experimentální část práce, neboť se jedná o značně netriviální úkol. K jeho obtížnosti přispěl i požadavek na čitelnost výsledných regulárních výrazů pro člověka.

Většinu vytyčených úkolů se podařilo splnit. Všechny navržené a implementované metody jsou v rámci nástroje YaraGen již v praxi používány ve společnosti Avast Software. Také došlo k úspěšné spolupráci se souběžně vypracovávanými bakalářskými pracemi ve společnosti Avast Software. Především se jedná o práci Jakuba Pružince [20], ve které přidal do nástroje Fileinfo generování několika nových rysů binárních souborů. Analýzy některých těchto rysů pak byly úspěšně zahrnuty i do systému YaraGen.

V případě generování regulárních výrazů se cíl podařilo naplnit jen částečně, proto by bylo vhodné se v budoucnu zaměřit na vylepšení tohoto algoritmu. Především bych rád zvýšil přehlednost generovaných výrazů. Dalším cílem do budoucna by mohlo být přidání regresních testů, které by tak doplnily existující integrační a jednotkové testy.

Kromě toho bych se v rámci další spolupráce se společností Avast Software rád zaměřil na dva významné úkoly, které by měly nástroj YaraGen zkvalitnit. Jedním z nich je rozlišování spolehlivosti jednotlivých hodnot na základě toho, v kolika procentech vstupních souborů se vyskytly. Druhým je vytvoření whitelistů jednotlivých rysů, aby nebyly do detekčních pravidel generovány generické hodnoty. Tím se omezí výskyt chyb prvního typu (*false positive*). V delším časovém horizontu je v plánu použít v nástroji YaraGen některé prvky umělé inteligence.

# Literatura


[1] AV-TEST: *The AV-TEST Security Report 2017/2018: The latest Analysis of the IT Threat Scenario*. [Online; navštíveno 6.01.2019].
URL https://www.av-test.org/en/news/the-av-test-security-report-20172018-the-latest-analysis-of-the-it-threat-scenario/

[2] BRIANDAIST, R. D. L.: File searching using variable length keys. In *IRE-AIEE-ACM '59 (Western)*, 1959.

[3] Chappell, G.: *The API Set Schema*. [Online; navštíveno 24.04.2019].
URL http://www.geoffchappell.com/studies/windows/win32/apisetschema/index.htm

[4] Cooper, D.; Santesson, S.; Farrell, S.; aj.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, RFC Editor, May 2008.
URL http://www.rfc-editor.org/rfc/rfc5280.txt

[5] Cuckoo: *Cuckoo Sandbox Book*. [Online; navštíveno 9.01.2019].
URL https://cuckoo.readthedocs.io

[6] tqdm developers: tqdm. [Online; navštíveno 02.05.2019].
URL https://tqdm.github.io/

[7] Eastlake, D.; Hansen, T.: US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634, RFC Editor, July 2006.
URL http://www.rfc-editor.org/rfc/rfc4634.txt

[8] Eastlake, D.; Jones, P.: US Secure Hash Algorithm 1 (SHA1). RFC 3174, RFC Editor, September 2001, http://www.rfc-editor.org/rfc/rfc3174.txt.
URL http://www.rfc-editor.org/rfc/rfc3174.txt

[9] ISO 14882:2017: Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, Prosinec 2017.

[10] Kaliski, B.: PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, RFC Editor, March 1998.
URL http://www.rfc-editor.org/rfc/rfc2315.txt

[11] Kim, D.; Kwon, B. J.; Dumitras, T.: Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI. In *ACM Conference on Computer and Communications Security*, 2017.

[12] Kitware: *CMake*. [Online; navštíveno 27.04.2019].
URL https://cmake.org/

[13] Mandiant: *Tracking Malware with Import Hashing*. [Online; navštíveno 27.02.2019].
URL https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html

[14] Microsoft: *Object Names*. [Online; navštíveno 24.04.2019].
URL https://docs.microsoft.com/en-us/windows/desktop/sync/object-names

[15] Microsoft: *PE Format*. [Online; navštíveno 6.01.2019].
URL https://docs.microsoft.com/cs-cz/windows/desktop/Debug/pe-format

[16] Microsoft: *Time Stamping Authenticode Signatures*. [Online; navštíveno 20.04.2019].
URL https://docs.microsoft.com/en-us/windows/desktop/seccrypto/time-stamping-authenticode-signatures

[17] Microsoft: *Windows Authenticode Portable Executable Signature Format*. [Online; navštíveno 20.04.2019].
URL https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx

[18] Milkovič, M.: *Systém pro detekci vzorů v binárních souborech*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2017, vedoucí práce Matula Peter.

[19] Pathria, P., R. K.; Beale: *Statistical mechanics (third edition)*. Academic Press, 2011.

[20] Pružinec, J.: *Extraction of static features from binary applications for malware analysis*. Diplomová práce, Brno University of Technology, Faculty of Information Technology, Brno, 2019.

[21] Revuz, D.: Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, ročník 92, 1992: s. 181–189.

[22] Rivest, R. L.: The MD5 Message-Digest Algorithm. RFC 1321, RFC Editor, April 1992.
URL http://www.rfc-editor.org/rfc/rfc1321.txt

[23] Software, A.: RetDec. [Online; navštíveno 14.05.2019].
URL https://github.com/avast/retdec

[24] StatCounter: *Desktop Operating System Market Share Worldwide*. [Online; navštíveno 6.01.2019].
URL http://gs.statcounter.com/os-market-share/desktop/worldwide

[25] Union, I. T.: *Introduction to ASN.1*. [Online; navštíveno 20.04.2019].
URL https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

[26] Union, I. T.: *ITU-T Recommendation X.690*. [Online; navštíveno 20.04.2019].
URL https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

[27] VirusTotal: *YARA in a nutshell*. [Online; navštíveno 7.01.2019].
URL http://virustotal.github.io/yara/

[28] Willems, D. C.: *Sandbox Evasion Techniques – Part 1.* [Online; navštíveno 6.01.2019]. URL https://www.vmray.com/cyber-security-blog/sandbox-evasion-techniques-part-1/

[29] YARA: *YARA's documentation.* [Online; navštíveno 7.01.2019]. URL https://yara.readthedocs.io

# Příloha A

# Obsah přiloženého DVD