VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRE AUTOMATIZOVANÝ ZBER, VYHODNOTENIE A AKTUALIZÁCIU YARA PRAVIDIEL

SYSTEM FOR AUTOMATIC COLLECTION, EVALUATION AND UPDATING OF YARA RULES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ KUČMA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. DOMINIKA REGÉCIOVÁ**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Kučma Tomáš**
Program: Informační technologie
Název: **Systém pro automatický sběr, vyhodnocení a aktualizaci YARA pravidel**
System for Automatic Collection, Evaluation, and Updating of YARA Rules
Kategorie: Bezpečnost

Zadání:

1. Studujte jazyk pro popis vzorů YARA a nástroje s nimi pracující. Zaměřte se na detaily jazyka z pohledu syntaxe a sémantiky.
2. Seznamte se s životním cyklem YARA pravidel ve společnosti Avast. Tj. jak pravidla vznikají, jak se testují a vyhodnocují, a jak se ukládají a aktualizují. Také zkoumejte nároky a standardy kladené na tato pravidla.
3. Identifikujte externí (z pohledu Avastu), volně dostupné, zdroje YARA pravidel a navrhněte systém pro jejich začlenění do YARA infrastruktury společnosti Avast. Řešení musí zaručit automatické sbírání, testování, vyhodnocení kvality, případné modifikace a uložení pravidel. Zajistěte asociaci a průběžnou aktualizaci vytvořených pravidel s původním zdrojem.
4. Implementujte systém navržený v bodě 3. Vhodně použijte principy SOA (Architektury orientované na služby), DevOps, a TDD (Vývoj řízený testy).
5. Použijte vytvořenou službu na reálné zdroje YARA pravidel vybrané v bodě 3.
6. Svoji práci zhodnoťte, diskutujte případné problémy a možný budoucí vývoj.

Literatura:

- P. Szor: The Art of Computer Virus Research and Defense, Addison-Wesley Professional (2005), ISBN 978-0321304544
- Recorded Future: The Threat Intelligence Handbook, CyberEdge Group (2018), ISBN 978-0999035467
- Dokumentace jazyka YARA (https://yara.readthedocs.io/en/latest/)
- Malpedia: https://malpedia.caad.fkie.fraunhofer.de/ (online)
- Interní dokumentace společnosti Avast
- Dále dle doporučení vedoucího či konzultanta

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních tří bodů a rozpracování bodu čtvrtého.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Regéciová Dominika, Ing.**
Konzultant: Matula Peter, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 14. října 2021

Abstrakt

Táto práca sa zaoberá vytvorením systému pre firmu Avast, ktorý bude umožňovať jednoducho integrovať verejne dostupné pravidlá pre nástroj YARA zo zvolených externých zdrojov, pre interné využitie Avastom. Pravidlá budú zbierané, vyhodnotené a pravidelne aktualizované. Práca poskytuje informácie o nástroji YARA a rovnomennom jazyku, s primárnym dôrazom na syntax a sémantiku, a o využívaní tohto nástroju v Avaste. Ďalej obsahuje samotný návrh systému a jeho implementáciu. Systém bol úspešne vytvorený, otestovaný a nakoniec nasadený v Avaste.

Abstract

This thesis deals with creating a system for the company Avast, which will allow to easily integrate publicly available rules for the tool YARA from chosen external sources, for internal use by Avast. Rules will be collected, evaluated, and regularly updated. The thesis provides information about tool YARA and the language of the same name, with a primary focus on its syntax and semantics, and about the usage of this tool in Avast. It also contains the design of the system and its implementation. The system was sucessfully created, tested, and finally deployed in Avast.

Kľúčové slová

YARA, YARA pravidlá, Avast, malvér, klasifikácia malvéru, identifikácia malvéru

Keywords

YARA, YARA rules, Avast, malware, malware classification, malware identification

Citácia

KUČMA, Tomáš. *Systém pre automatizovaný zber, vyhodnotenie a aktualizáciu YARA pravidiel*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Dominika Regéciová

Systém pre automatizovaný zber, vyhodnotenie a aktualizáciu YARA pravidiel

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pani Ing. Dominiky Regéciovej. Ďalšie informácie mi poskytli Ing. Peter Matula a Ing. Marek Milkovič. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Tomáš Kučma
8. mája 2022

Poďakovanie

Rád by som sa poďakoval vedúcej Ing. Dominike Regéciovej, ako aj konzultantom z Avastu Ing. Peterovi Matulovi a Ing. Marekovi Milkovičovi, za podporu, vedenie a rady v súvislosti s touto bakalárskou prácou.

Obsah

Kapitola 1

Úvod

V dnešnom svete sú technológie čoraz väčšou súčasťou našich životov. Prenikajú do všetkých sfér spoločnosti a neustále sa vyvíjajú. Toto má svoje prínosy, avšak otvára to aj dvere novým problémom. Kyberkriminálnici sa pokúšajú odhaliť slabiny technológií, ktoré by mohli využiť vo svoj prospech, čo vedie ku škodám pre legitímnych užívateľov. Proti takýmto hrozbám nás pomáhajú chrániť odborníci pracujúci v oblasti počítačovej bezpečnosti. Na strane kyberkliminality nastávajú konštantné pokroky a nové snahy o prelomenie bezpečnosti. Preto je nutné, aby aj ich protistrana neustále napredovala.

Existuje mnoho rôznych nástrojov využívaných na boj proti počítačovým hrozbám. Jedným z takýchto nástrojov je YARA. YARA používa špeciálny jazyk, pomocou ktorého môžeme vytvárať popisy, takzvané pravidlá, ktoré nám v kombinácií so samotným nástrojom umožňujú identifikovať a klasifikovať vzorky malvéru. Tieto pravidlá sa dajú jednoducho zdieľať a týmto sa zvyšuje schopnosť pohotovo reagovať na novo objavujúce sa hrozby.

Táto práca vzniká v spolupráci s firmou Avast. Avast je jednou z firiem pracujúcich na zabezpečení počítačových technológií. Ako v mnoho iných firmách pracujúcich v oblasti počítačovej bezpečnosti, aj tu je využívaný nástroj YARA.

Cieľom tejto práce je vytvoriť systém pre malvérových analytikov Avastu, ktorý im zjednoduší získavanie a využívanie voľne dostupných YARA pravidiel. Tento systém zautomatizuje prácu, ktorá by sa inak musela vykonávať manuálne, čím sa zvýši produktivita a spoľahlivosť. Bude zaistené nie len uľahčenie využívania daných pravidiel, ale taktiež ich kvalita a aktuálnosť.

Táto bakalárska práca obsahuje úvod do danej problematiky, ako vo všeobecnom kontexte, tak aj v kontexte Avastu, následne návrh samotného systému, jeho implementáciu a nakoniec aj testovanie. Celkovo teda má 5 obsahových kapitol, plus úvod a záver. Kapitola číslo 2 obsahuje informácie týkajúce sa YARY. Vysvetľuje princípy samotného nástroja ako aj využívaného jazyka, s primárnym zameraním na syntax a sémantiku. V ďalšej kapitole, číslo 3, sú poskytnuté informácie o využívaní YARY vo firme Avast, vrátane pracovných postupov a systémov s ňou súvisiacich. Kapitola venovaná návrhu, čiže kapitola 4, vysvetľuje návrh systému ako celku a návrh jednotlivých častí systému. 5. kapitola obsahuje podrobný popis implementácie vychádzajúci z návrhu. Testovanie v rámci vývoja ako aj hotového finálneho systému a výsledky tohto testovania sú spracované v kapitole číslo 6.

Kapitola 2

YARA

YARA je nástroj primárne zameraný na uľahčovanie identifikácie a klasifikácie malvérových vzoriek. Názov YARA sa taktiež vzťahuje na jazyk používaný na tvorbu detekčných pravidiel, na základe ktorých pracuje spomínaný nástroj.

Na začiatku tejto kapitoly sa nachádza úvod do malvéru a jeho klasifikácie. Následnej je vysvetlený a popísaný jazyk YARA, ako aj rovnomenný nástroj. Kapitola čerpá primárne zo zdroja [17]. Tento zdroj sa vzťahuje na verziu YARY 4.1.3.

2.1 Malvér a jeho klasifikácia

Označenie *malware* (slovensky *malvér*) vzniklo ako spojenie slov *malicious software*, čiže škodlivý softvér. Pod tým sa rozumie akýkoľvek softvér, ktorý spôsobuje škodu užívateľovi, počítaču alebo sieti [15].

Podľa zdroja [16] sú základné kategórie malvéru nasledovné:

- *Viruses* (vírusy) — vírus je kód, ktorý rekurzívne replikuje potenciálne vyvinutú kópiu seba samého.
- *Worms* (červy) — sú to sieťové vírusy, replikujúce sa primárne na sieti. Zvyčajne sa šíria na ďalšie stroje bez zásahu užívateľa, avšak niekedy je uživateľ potrebný na daľšie šírenie.
- *Logic bombs* (logické bomby) — logická bomba označuje naprogramovanú poruchu legitímnej aplikácie. Príkladom môže byť vymazanie softvéru po určitom počte spustení.
- *Trojan horses* (Trójske kone) — trojský kôň je malvérový program, ktorý sa snaží zaujať užívateľov nejakou ponúkanou funkcionalitou, pričom jeho skutočným účelom je vykonať nejakú škodlivú činnosť.
- *Germs* — sú to vírusy prvej generácie vo forme, ktorú vírus nedokáže generovať v rámci jeho bežných infekčných procesov.
- *Exploits* — *exploit* je kód špecifický pre nejakú slabinu či sadu slabín. Môže byť využitý napríklad na získanie vysoko privilegovaného prístupu na cieľový systém.
- *Downloaders* — tento malvér na zariadenie stiahne sadu iných predmetov, napríklad ďalší malvér, ktorý aj spustí.

- *Dialers* — začali byť používané za čias vytáčaného pripojenia. Spôsobovali, že bez vedomia užívateľa jeho zariadenie vytáčalo čísla platených telefónnych liniek.
- *Droppers* — tento výraz označuje inštalátory pre vírus prvej generácie. *Dropper* môže napríklad uložiť *germ* kód do zavádzacieho sektoru diskety.
- *Injectors* — sú špeciálnym druhom malvéru *dropper*, vírusový kód inštalujú do pamäte.
- *Auto-Rooters* — zvyčajne nástroje hekerov, používané na získanie prístupu do nových zariadení zo vzdialených strojov.
- *Kits* — taktiež známe ako generátory vírusov. Aplikácie, ktoré aj neskúseným užívateľom umožňujú jednoducho vytvárať nové počítačové vírusy.
- *Spammer programs* — sú používané na posielanie nevyžiadaných správ cez e-mail, SMS, atď.
- *Flooders* — sú používané na „zaplavenie“ počítčovej siete, čiže na vykonávanie DoS — denial of service (zamietnutie služby) — útokov.
- *Keyloggers* — sú používané na zachytávanie stlačených kláves na infikovanom systéme, vďaka čomu útočníci získavajú prístup k citlivým informáciám, ako napr. heslám alebo číslam kreditných kariet.
- *Rootkits* — špeciálne sady hekerských nástrojov, ktoré sa používajú až po tom, čo sa útočník získal koreňový prístup do počítačového systému.

Ďalej uvádza, že je malvér možné bližšie klasifikovať podla jeho malvérovej rodiny, variantu, cieľovej platformy, atď.

Výraz malvérová analýza označuje prácou s malvérom za cieľom pochopenia toho, ako tento softvér funguje, ako sa dá identifikovať a ako sa dá poraziť alebo eliminovať. Existujú dva typy malvérovej analýzy, statická a dynamická. Statická analýza je vykonávaná bez spúšťania vírusovej vzorky. Na druhú stranu, dynamická analýza spočíva na princípe spúšťania malvéru a sledovania jeho správania. Pre vykonávanie dynamickej analýzy je potrebné využívať špeciálne prostredie, aby nebola ohrozená sieť alebo počítač osoby vykonávajúcej analýzu [15].

2.2 Jazyk YARA

Jazyk YARA definuje syntaktickú a sémantickú štruktúru YARA pravidiel. Podmienka pravidla definuje, pre ktoré súbory alebo procesy nastáva zhoda s daným pravidlom. YARA pravidlá sa zoskupujú do súborov. Okrem definícií pravidiel môže súbor obsahovať importovanie modelu (kľúčové slovo `import` nasledované názvom modulu v úvodzovkách) a vloženie iného súboru (kľúčové slovo `include` nasledované cestou k súboru v úvodzovkách).

Jednotlivé pravidlá sa skladajú s viacerých častí. Všeobecná štruktúra pravidla je uvedená na výpise 2.1. Z vyznačených častí sú povinné iba kľúčové slovo `rule`, množinové zátvorky, časť `<name>` a časť `<condition>`. Zvyšné časti môžu byť prázdne. Pre identifikátory všeobecne platia rovnaké pravidlá ako v jazyku C, čiže môžu obsahovať iba alfanumerické znaky a podčiarkovník, nesmú začínať číslom, a rozlišujú sa malé a veľké písmená. YARA tiež podporuje jednoriadkové a viacriadkové komentáre, zapísané rovnako ako v jazyku C (`//...` alebo `/*...*/`).

```
<rule_modifiers> rule <name> <tags> {
    <meta>
    <strings>
    <condition>
}
```

Výpis 2.1: Všeobecná štruktúra YARA pravidla

Časť `<rule_modifiers>`

Táto časť definuje modifikátory pravidla. Pre pravidlá existujú dva modifikátory, ktoré je možné kombinovať. Modifikátor `global` umožňuje tvorbu obmedzení pre všetky pravidlá zároveň. Aby mohla nastať úspešná zhoda pre akékoľvek iné pravidlo, musí byť podmienka pravidla s modifikátorom `global` splnená. Modifikátor `private` určuje, že zhody s modifikovaným pravidlom sa nevypisujú na výstup. Používa sa primárne pre pravidlá, ktoré sú určené na kombináciu s inými pravidlami (modifikátorom `global` alebo iným spôsobom).

Časť `<name>`

Táto časť obsahuje názov pravidla, ktorý sa využíva sa pri výpise alebo pri odkazovaní sa na dané pravidlo. Pre názov pravidla platia rovnaké pravidlá ako pre identifikátory a musí byť unikátny naprieč všetkými pravidlami súboru.

Časť `<tags>`

Táto časť slúži na označenie pravidla značkami, pomocou ktorých je možné pravidlá zoskupovať a následne sa na takto vytvorené skupiny odkazovať. Táto časť začína znakom „:“, nasleduje neprázdny zoznam značiek. Značky v rámci zoznamu musia byť unikátne.

Časť `<meta>`

Táto časť má čisto informačný význam, nemá žiaden vplyv na sémantiku pravidla. Začína kľúčovým slovom `meta`, po čom nasleduje znak „:“ a neprázdny zoznam meta informácií. Meta informácia sa skladá z identifikátora a hodnoty, oddelených znakom „=“. Hodnotou môže byť reťazec, integer alebo booleovská hodnota, zapísané rovnako ako v jazyku C.

Časť `<strings>`

Táto časť umožňuje definovať reťazce, ktoré sú používané v podmienke pravidla na detekciu ich hodnoty vo spracovávanej vzorke. Začína kľúčovým slovom `strings`, po čom nasleduje znak „:“ a neprázdny zoznam reťazcov. Pre každý reťazec je zadefinovaný unikátny identifikátor, samotná hodnota reťazca (oddelená od identifikátora znakom „=“), a nepovinný zoznam modifikátorov reťazca. Identifikátory reťazca musia začínať znakom „$“. Vytvorením identifikátora pozostávajúceho iba zo znaku „$“ sa definuje anonymný reťazec (ktorých môže byť aj viacero), na ktorý sa nedá odkazovať priamo. Podľa hodnoty reťazca existujú tri typy reťazcov:

- **Textové reťazce** — jednoduché textové ASCII reťazce. Sú ohraničené znakom dvojitých úvodzoviek. Okrem textu podporujú niektoré tzv. escape sekvencie jazyka C: `\"`, `\\`, `\r`, `\t`, `\n`, `\xdd`. Príkladom definície textového reťazca je:

```
$text = "Prvy riadok\nDruhy riadok"
```

- **Hexadecimálne reťazce** — používajú sa na definovanie sekvencie bajtov. Sú ohraničené znakmi „{“ a „}“. Hodnoty jednotlivých bajtov sú zapisované hexadecimálne. Tieto reťazce umožňujú využívať špeciálne konštrukcie. Znak „?“ v čísle značí ľubovolnú 4 bitovú hodnotu. Znak „|“, v kombinácií zo zátvorkami, umožňuje vytvárať alternatívne sekvencie bajtov. Konštrukcia `[X-Y]` označuje skok ľubovolný počet bajtov v danom rozsahu (X je implicitne 0 a Y nekonečno). Príkladom definície hexadecimálneho reťazca je:

```
$hex = { 4F B? ( FF | ?E EE ) [5-] 0E }
```

- **Regulárne výrazy** — sú ohraničené znakom „/“. Platia pre ne rovnaké pravidlá ako pre regulárne výrazy v jazyku Perl[1], s niektorými výnimkami, ako je napríklad absencia podpory pre zachytávajúce skupiny (capture groups), POSIXové triedy znakov (POSIX character classes) a spätné odkazy (backreferences). Príkladom definície regulárneho výrazu je:

```
$regex = /[a-zA-Z]+\.(sk|cz)/
```

Ako bolo spomínané, za hodnotu reťazca je možné pridať zoznam modifikátorov. Dostupné modifikátory závisia od typu reťazca, a v niektorých prípadoch sa nesmú kombinovať. Kompletný zoznam modifikátorov je:

- `nocase` — nezáleží na veľkosti písmen.
- `wide` — po každom znaku je vložený null (0x00) znak pre emuláciu UTF16.
- `ascii` — hľadá sa zhoda aj s ASCII znakmi, potrebné iba pri použití `wide`.
- `xor` — hľadá sa zhoda aj s akýmkoľvek reťazcom, ktorý vznikne aplikovaním XOR operácie (s ľubovolne zvolenou konštantou) na každý znaky reťazca.
- `base64` — konvertuje reťazec na base64 kódovanie (umožňuje tiež definíciu vlastnej base64 abecedy 64 znakovým reťazcom v zátvorke za kľúčovým slovom).
- `base64wide` — rovnaké správanie ako `base64`, ale výsledok sa konvertuje na 16 bitový znak
- `fullword` — reťazec nemôže byť prechádzaný ani nasledovaný alfanumerickým znakom.
- `private` — reťazec nebude v žiadnom prípade zahrnutý vo výstupe YARA.

[1] http://www.pcre.org/

Modifikátor	Povolený typ reťazca	Nemožno kombinovať s
`nocase`	text, regulárny výraz	`xor`, `base64`, `base64wide`
`wide`	text, regulárny výraz	—
`ascii`	text, regulárny výraz	—
`xor`	text	`nocase`, `base64`, `base64wide`
`base64`	text	`nocase`, `xor`, `fullword`
`base64wide`	text	`nocase`, `xor`, `fullword`
`fullword`	text, regulárny výraz	`base64`, `base64wide`
`private`	hexadecimálny, text, regulárny výraz	—

Obr. 2.1: Tabuľka obmedzení pre modifikátory

Reťazce s modifikátormi potom môžu vyzerať napríklad takto:

```
$text = "Prvy riadok\nDruhy riadok" xor
$hex = { 4F B? ( FF | ?E ee ) [5-] 0E } private
$regex = /[a-zA-Z]+\.(sk|cz)/ ascii wide
```

Časť `<condition>`

Táto časť obsahuje definíciu podmienky pravidla. Podmienka má formu logického výrazu, ktorého pravdivostná hodnota určuje, či nastala zhoda pre spracovávanú vzorku. Táto časť začína kľúčovým slovom `condition`, po čom nasleduje znak „:“, a samotný výraz podmienky. Pre pravidlá sú definované následujúce druhy literálov:

- Integer literál (v dekadickej podobe alebo s prefixom `0x` v hexadecimálnej podobe).
- Literál čísla s plávajúcou čiarkou.
- Reťazcový literál (textový reťazec alebo regulárny výraz).

Sú definované nasledujúce druhy operátorov:

- Logické operátory pre booleovské typy `and`, `or` a `not`.
- Aritmetické operátory pre číselné typy `+`, `-` (unárne aj binárne) , `*`, `\` (celočíselné delenie) a `%`.
- Bitové operátory pre typ integer `&`, `|`, `<<`, `>>`, `~` a `^`.
- Relačné operátory pre číselné typy a textové reťazce `>=`, `<=`, `<`, `>`, `==` a `!=`.
- Relačné operátory pre textové reťazce `contains`, `startswith`, `endswith` a ich varianty `icontains`, `istartswith`, `iendswith` — test či textový reťazec vľavo obsahuje, obsahuje na začiatku alebo obsahuje na konci textový reťazec vpravo. Varianty s prefixom `i` ignorujú veľkosť písma.
- Relačný operátor `matches` — test či textový reťazec vľavo je popísaný regulárnym výrazom vpravo.

S reťazcami zadefinovanými v sekcií `<strings>` sú definované nasledujúce spôsoby práce:

- `$<string-identifier>` (pričom `<string-identifier>` je identifikátor reťazca bez prvého znaku `$`) — nadobúda hodnotu `true`, ak nastala zhoda s reťazcom, inak `false`.
- `$<string-identifier> at <offset>` (pričom `<offset>` je typu integer) — nadobúda hodnotu `true`, ak nastala zhoda s reťazcom na ofsete danej hodnoty, inak `false`.
- `$<string-identifier> in (<N>..<M>)` (pričom `<N>` a `<M>` sú typu integer) — nadobúda hodnotu `true`, ak nastala zhoda s reťazcom na ofsete v rozsahu od N po M, inak `false`.
- `#<string-identifier>` — nadobúda hodnotu počtu výskytov daného reťazca.
- `@<string-identifier>` a `@<string-identifier>[<N>]` — nadobúda hodnotu ofsetu N-tého výskytu reťazca. N je implicitne 1. Ak daný výskyt neexistuje, nadobúda hodnotu NaN — Not A Number.
- `!<string-identifier>` a `!<string-identifier>[<N>]` — nadobúda hodnotu dĺžky N-tého výskytu reťazca. N je implicitne 1. Ak daný výskyt neexistuje, nadobúda hodnotu NaN — Not A Number.

Ďalej je možné využívať cykly so všeobecným tvarom

```
for <amount> <id> <keyword> <iterable> : (<condition>)
```

Iteruje sa cez množinu `<iterable>` a pre každú iteráciu sa vyhodnotí definovaná podmienka `<condition>`. Hodnota `<amount>` určuje minimálny počet iterácií, ktoré musia byť vyhodnotené ako pravdivé, aby bol cyklus vyhodnotený ako pravdivý. Táto hodnota môže byť integer, kľúčové slovo `any` (ekvivalent čísla 1), alebo kľúčové slovo `all` (ekvivalent počtu všetkých iterácií). Existujú viaceré typy cyklov:

- Reťazcový cyklus — časť `<id>` je vynechaná. `<keyword>` je kľúčové slovo `of`. Množina `<iterable>` je uzátvorkovaný zoznam identifikátorov reťazcov oddelených čiarkou. Identifikátory môžu aj obsahovať zástupný znak `*`, kedy identifikátor reprezentuje všetky definované reťazcové identifikátory získateľné nahradením daného znaku ľubovolným počtom ľubovolných znakov. Množina `<iterable>` môže byť tiež nahradená kľúčovým slovom `them`, ktoré sa odkazuje na všetky zadefinované reťazce (ekvivalent výrazu `($*)`). Na reťazec aktuálnej iterácie sa v podmienke odkazuje pomocou výrazov s prázdnym `<string-identifier>` (napr. `$`, `@[1]`, ...). Cyklus s podmienkou ekvivalentnou výrazu `$` môže byť zjednodušený na `<amount> of <iterable>` (napr. `any of them`).
- Ostaté typy cyklov — pole `<id>` obsahuje identifikátor, pomocou ktorého sa v podmienke odkazuje na element aktuálnej iterácie. Pre slovník obsahuje dva identifikátory oddelené čiarkou, pre kľúč a hodnotu. `<keyword>` je kľúčové slovo `in`. `<iterable>` môže byť poľom, slovníkom, rozsahom `(<N>..<M>)`, alebo zoznamom uzátvorkovaných integerov oddelených čiarkou.

Ďalej je možné priamo pristupovať ku pamäti cieľa pomocou `int8(<offset>)`. Existuje aj 16 a 32 bitová varianta, a je možné pridať prefix `u` pre bezznamienkové hodnoty.

Identifikátor iného pravidla, ak je použitý v podmienke, nadobúda pravdivostnú hodnotu na základe vyhodnotenia daného pravidla.

K importovaným modulom sa pristupuje pomocou identifikátoru daného modulu. Modul má formu štruktúry. K prvkom štruktúr sa pristupuje operátorom `.<member>` (kde `<member>` je názov prvku), k prvkom poľa alebo slovníku sa pristupuje operátorom `[<key>]` (kde `<key>` je kľúč alebo index). Funkcie sa taktiež volajú rovnakým spôsobom ako v jazyku C.

Kľúčové slovo `filesize` nadobúda hodnotu veľkosti súboru (ak ide o súbor). Kľúčové slovo `entrypoint` nadobúda hodnotu hlavného vstupného bodu programu (ak má podporovaný formát alebo ide o proces).

2.3 Príklad pravidla YARA

```
rule example : tag {
    meta:
        author = "Tomas Kucma"
    strings:
        $str1 = "prvy retazec" nocase
        $str2 = "druhy retazec"
        $hex = { E7 4B 00 FF }
        $regex = /[\d]+\. retazec/
    condition:
        all of ($str*, $hex) or $regex
}
```

Výpis 2.2: Príklad jednoduchého YARA pravidla

Jednoduché YARA pravidlo možno vidieť na výpise 2.2. Toto pravidlo má názov `example` a značku `tag`. V časti `meta` je informácia o autorovi pravidla. Ďalej nasleduje časť `strings` definujúca štyri reťazce. Reťazce `$str1` a `$str2` definujú jednoduché ASCII reťazce, pričom v prvom prípade nezáleží na veľkosti písmen. `$hex` definuje postupnosť bajtov. `$regex` definuje regulárny výraz. Samotná podmienka v časti `condition` definuje, že zhoda s pravidlom nastane ak sa v skenovanom súbore nachádzajú oba textové reťazce a hex reťazec, alebo ak nastala zhoda s regulárnym výrazom.

2.4 YARA moduly

YARA umožňuje rozšírenie funkcionality pomocou modulov. S využitím modulov sa dá zadefinovať nové dátové štruktúry a funkcie, ktoré možno použiť na tvorbu zložitejších podmienok. Existuje niekoľko modulov, ktoré sú súčasťou oficiálnej distribúcie YARA:

- Modul `PE` — tento modul tvorbu zložitejších pravidiel pre súbory vo formáte PE. PE je formát spustiteľných súborov vyvíjaný spoločnosťou Microsoft, používaný na platforme Windows. S pomocou modulu je možné pristupovať k hlavičkám daného formátu.
- Modul `ELF` — tento modul tvorbu zložitejších pravidiel pre súbory vo formáte ELF. Podobne ako PE, ELF je formát spustiteľných súborov, avšak primárne používaný

na Unixových a Unix-like platformách. Existujú 3 hlavné typy súborov tohto formátu: spustiteľný súbor, objektový súbor obsahujúci dáta na linkovanie z ostatnými objektovými súbormi a dynamická knižnica [13].

- Modul `Cuckoo` — tento modul umožňuje tvorbu pravidiel využívajúcich informácie o správaní programu generovaných pomocou Cuckoo Sandbox. Cuckoo Sandbox je nástroj, ktorý sa zameriava na dynamickú analýzu malvéru. Na základe správania malvéru v izolovanom prostredí je vygenerovaná správa [10].
- Modul `Magic` — tento modul umožňuje identifikovať typ súboru na základe výstupu unixového príkazu `file`. Na platforme Windows tento modul nie je podporovaný.
- Modul `Hash` — tento modul poskytuje funkcie na vypočítavanie hešov (MD5, SHA1, SHA256) pre súbory či časti súborov.
- Modul `Math` — tento modul poskytuje funkcie pre niektoré matematické výpočty, ako napríklad výpočet entropie, odchýlky, minima, či maxima.
- Modul `Dotnet` — tento modul umožňuje vytvárať presnejšie pravidlá pre .NET súbory na základe atribút a vlastností .NET súborového formátu.
- Modul `Time` — tento modul poskytuje funkciu pre získanie aktuálneho času čím umožňuje tvorbu časovo závislých.

YARA taktiež umožňuje tvorbu vlastných modulov. Moduly sa tvoria v jazyku C a musia byť do nástroja YARA zahrnuté už počas kompilácie.

Príklad použitia modulu možno vidieť na výpise 2.3.

```
import "pe"

rule PE_rule {
    meta:
        author = "Tomas Kucma"
        pe = true
    strings:
        $str = "BEGIN"
    condition:
        $str at pe.entry_point and pe.number_of_sections == 2
}
```

Výpis 2.3: Príklad textu súboru obsahujúceho pravidlo využívajúce modul `PE`

2.5 Nástroj YARA

YARA je multiplatformový (Windows, Linux, Mac OS X) nástroj. Na základe YARA pravidiel, popisujúcich vlastnosti, vyhľadáva vo cieľoch zhody [11].

Existujú viaceré možné spôsoby používania nástroja YARA. Základný spôsob je spúšťanie YARA z príkazového riadku. V takom prípade sa využíva nasledovne:

```
$ yara [OPTIONS] RULES_FILE TARGET
```

kde `RULES_FILE` označuje súbor obsahujúci pravidlá, `TARGET` označuje cieľový súbor alebo adresár a `OPTIONS` ďalšie voliteľné možnosti ako napríklad rekurzívne prechádzanie priečinkov. Pri štandardnom spustení YARA na výstup vypíše zoznam súborov a pravidiel medzi ktorými nastala zhoda. Môže to vyzerať napríklad takto:

```
$ yara rules.yar .
rule_A ./example.yar
rule_B ./example.yar
rule_B ./test.txt
rule_A ./rules.yar
```

Súčasťou YARA je aj nástroj `yarac`, ktorý umožňuje pravidlá skompilovať do binárnej podoby. Súbory pravidiel v binárnej podobe majú zvyčajne niekoľkonásobnú veľkosť ako pravidlá v podobe textovej, avšak umožňujú nástroju YARA pracovať rýchlejšie pri samotnej detekcií [7].

Nástroj tiež existuje vo forme C knižnice `libyara`, ktorá poskytuje API — application programming interface (aplikačné programovacie rozhranie). Jedná sa o rovnaké API využívané nástrojom YARA príkazového riadku. YARA sa dá tiež využívať v programovacom jazyku Python pomocou knižnice `yara-python`.

Kapitola 3

YARA v Avaste

Rovnako ako mnoho iných antivírusových spoločností, aj Avast využíva YARU. Na YARE sú závislé viaceré systémy Avastu. Všetky YARA pravidlá sú zoskupované v jednom centrálnom úložisku.

V tejto kapitole sú popísané relevantné informácie ohľadom využívania YARY v Avaste. V prvej podkapitole sú uvedené štandardy kladené na pravidlá. Ďalej sa tu nachádza podkapitola uvádzajúca vybrané technológie používané Avastom, ktoré súvisia s využitím YARY. Nakoniec je popísaný životný cyklus pravidiel. Informácie sú čerpané primárne z interného Avastieho zdroja [5], s výnimkou podkapitoly o technológiách.

3.1 Štandardy pre YARA pravidlá v Avaste

Avast vyžaduje, aby boli YARA pravidlá v určitom formáte a obsahovali určité informácie. V tejto podkapitole sú popísané všetky podstatné požiadavky Avastu.

Kategórie pravidiel

Pravidlá sú rozdelené do troch hlavných kategórií:

- `classification` — pravidlá pre klasifikáciu špecifických malvérových rodín.
- `hunting` — pravidlá, ktoré nie sú špecifický zamerané na konkrétnu malvérovú rodinu, ale identifikujú všeobecne podozrivé správanie či sekvencie. Tieto pravidlá majú v názve prefix `hunting_`.
- `includes` — privátne pravidlá, ktoré sú využívané inými pravidlami.

Adresárová štruktúra úložiska pravidiel sa riadi podľa kategórií pravidiel. Pre kategórie `classification` a `hunting` sú na druhej úrovni adresárovej štruktúry pravidlá ďalej zoskupené do adresárov podľa bežných systémov: `win`, `linux`, `misc`, atď. Pravidlá typu `classification` sú v ďalších úrovniach ešte delené podľa závažnosti a následne typu. Pravidlá kategórie `includes` sú uložené buď priamo v rovnomennom adresári alebo v jeho podadresári `hunting`.

Druhy pravidiel

Klasifikačné YARA pravidlá sa v Avaste delia behaviorálne pravidlá a statické pravidlá. Podľa druhu pravidiel sa k názvu pravidiel dáva sufix:

- `_known_named_objects` — pravidlo využívajúce detekciu podľa synchronizačných prostriedkov malvéru (semafory, rúry, udalosti, atď.). Zvyčajne sú to najspoľahlivejšie pravidlá.
- `_known_behavior_high` — pravidlo založené na správaní (zmenené súbory, navštívené URL, atď.) s vysokou spoľahlivosťou.
- `_known_behavior_low` — pravidlo založené na správaní s nízkou spoľahlivosťou.
- `_known_sequences` — pravidlo založené na statických vlastnostiach (obsah hlavičky, sekvencie bajtov, atď.).

Názvy privátnych pravidiel sa zapisujú veľkým písmom, názvy ostatných pravidiel malým písmom.

Metadáta

Informácie o konkrétnych významoch a možných hodnotách jednotlivých meta informácií boli doplnené zo zdroja [7].

Každé pravidlo by malo mať nasledujúce informácie:

- `author` (autor) — meno autora, plus „, Avast“ ak je z Avastu.
- `description` (popis) — krátky popis pravidla.
- `hash` — SHA256[1] heš súboru použitého na tvorbu pravidla. Táto informácia sa môže vyskytovať viac krát, pre každý z použitých súborov. Pre pravidlá, pre ktoré táto informácia nie je relevantná, nie je uvedená.
- `rule_type` (typ pravidla) - môže byť `static` (statické), `behavioral` (behaviorálne), `mixed` (zmiešané) alebo `other` (iné).

Pre klasifikačné pravidlá sa pridávajú informácie:

- `severity` (závažnosť) — môže byť `malware` (malvér), `pup` (potentially unwanted application — potenciálne nechcená aplikácia), `tool` (nástroj) alebo `unknown` (neznáma).
- `reliability` (spoľahlivosť) — môže byť `kill` (zabiť), `report` (nahlásiť), `susp` (podozrivé) alebo `test`. Momentálne nie je rozdiel medzi dvoma prvými úrovňami, obe implikujú automatickú klasifikáciu, narozdiel od zvyšných dvoch. `susp` sa používa, keď je žiadúce nechať pravidlo v neklasifikujúcom stave. `test` je predvolený stav pre novo pridané pravidlá.
- `type` (typ) — typ malvéru zapísaný malými abecednými znakmi, prípadne s pomlčkou. Napríklad `ransomware`, `worm`, `trojan`, ...
- `strain` (rodina malvéru) — konkrétna rodina malvéru, napr. `WannaCry`, `CrySiS`, ...

Pre `hunting` pravidlá sa pridáva informácia `hunting_tag`, ktorý identifikuje sledovanú vlastnosť.

Môžu byť použité aj ďalšie meta informácie, ktoré sú ale nepovinné.

[1] https://datatracker.ietf.org/doc/html/rfc6234

Reťazce

Reťazce sa zvyčajne nazývajú podľa konvencie:

- `$s00-$s99` — textové reťazce.
- `$h00-$h99` — hexadecimálne reťazce.
- `$r00-$r99` — regulárne výrazy.
- `$b00-$b99` — akýkoľvek typ reťazca používaný na zamedzenie zhody (blacklisting).

Podmienky

Statické a behaviorálne podmienky by sa nemali kombinovať, ak to nie je nutné. Jednotlivé časti podmienok (napr. oddelené operátorom `and` alebo `or`) sú každá na novom riadku. Pre indentáciu sa využíva tab.

3.2 Vybrané technológie používané Avastom

V tejto podkapitole sú popísané podstatné technológie, ktoré sú v Avaste využívané, týkajúce sa tejto práce. Ide o systém Git, službu GitHub, aplikáciu RabbitMQ a Avastí systém YARKA.

Git a GitHub

Git[6] je nástroj typu VSC — version control system (systém správy verzií). Takéto nástroje slúžia na správu a prácu s rôznymi verziami softvéru, prípadne iného obsahu. Pre prácu so systémom Git je podstatné poznať nasledujúce termíny a koncepty:

- *Repository* (repozitár) — databáza obsahujúca všetky informácie potrebné na udržovanie a správu revízií a histórie projektu. V repozitári, Git udržuje dve primárne dátové štruktúry *object store* (úložisko objektov) a *index*.
- *Object store* — obsahuje Git objekty. Pre každý objekt existuje je vygenerovaný heš, ktorý je jeho *object ID* (ID objektu). Pre git objekty existujú 4 atomické typy:
  - *Blob* (Binary large object, slovensky „binárny veľký objekt“) — obsahuje dáta súborov, avšak nie ich metadáta. Z hľadiska systému Git nemá internú štruktúru.
  - *Tree* (strom) — reprezentuje jeden priečinok. Udržuje identifikátory blobov, názvy ciest a časť metadát pre súbory v danom priečinku. Taktiež môže obsahovať referencie na podstromy, vďaka čomu vzniká kompletná hierarchia priečinkov a súborov.
  - *Commit* — udržuje metadáta pre každú zmenu aplikovanú v repozitári, čiže autora, dátum commitu, správa, atď. Okrem toho odkazuje na strom, ktorý zachytáva stav repozitára v momente commitu. Koreňový commit nemá rodiča, avšak ostatné commity áno. Rodičom je commit, na ktorý tento `commit` naväzuje. V špeciálnych prípadoch môže mať commit aj viacero rodičov.
  - *Tag* (značka) — priraďuje ľubovoľné, zvyčajne pre ľudí čitateľné, meno špecifickému objektu.

- *Index* — dočasný dynamický súbor, ktorý zachytáva verziu celkovej štruktúry projektu v nejakom časovom bode. Do indexu sa pomocou Git príkazov môžu pridávať zmeny vykonané na súboroch. Následne je možné príkazom commit vytvorí nový objekt commit odrážajúci tieto zmeny.

- *Branch* (vetva) — commity, ktoré na seba naväzujú, tvoria vetvu. Všetky commity v systéme Git patria nejakej vetve (prípadne viacerým). Existencia viacerích vetiev umožnuje, aby sa repozitár rozvetvoval. Hlavná vetva za zvyčajne nazýva `master` alebo `main`.

- *Merge* (zlúčenie) — aplikovanie zmien z jednej vetvy na druhú. Výsledkom je commit, ktorý má dvoch rodičov. V prípadoch, že zmeny jednej a druhej vetvy vykonané od posledného spoločného commitu sa vzájomne vylučujú, nastáva *merge conflict*. Vtedy sa vyžaduje manuálny zásah, ktorý označí, ako ma vyzerať výsledok zlúčenia.

GitHub je služba poskytujúca hosting Git repozitárov. Okrem svojej hlavnej, verejne dostupnej domény github.com, ktorú je možno využívať bezplatne (s obmedzeniami), ponúka aj služby špeciálne pre firmy. Tento produkt sa volá GitHub Enterprise, a umožňuje využívanie rozhrania a funkcionality služby GitHub, pričom dáta sú udržované bezpečne na vlastných serveroch danej firmy [3]. Produkt GitHub Enterprise je využívaný aj firmou Avast.

GitHub navyše poskytuje rozšírenú funkcionalitu a možnosti práce s Git repozitármi [2]. Jednou z najpodstatnejších funkcií je možnosť vytvoriť PR (*pull request*), čo je vlastne žiadosť na merge z jednej vetvy na druhú. PR slúžia na zjednodušenie kolaboratívneho vývoja. Zmeny sa primárne vykonávajú inkrementálne na vedľajších vetvách a až po dokončení uceleného celku sa pre tieto zmeny vytvorí PR s hlavnou vetvou ako cieľom. PR ponúka prehľad zmien a umožňuje na ne reagovať aj ostatným ľudom využívajúcim repozitár. Na základe spätnej väzby od ostatných ľudí sa následne môžu vykonať nejaké doplňujúce úpravy alebo sa PR môže schváliť a vykoná sa `merge`. GitHub taktiež poskytuje rozhranie na riešenie prípadných konfliktov.

RabbitMQ

RabbitMQ je *message broker*, čiže aplikácia, ktorá sprostredkúva komunikáciu pomocou správ medzi rôznymi aplikáciami. RabbitMQ bol pôvodne vytvorený pre AMQP protokol, z ktorého vychádza, aj keď neskôr bol rozšírený pre podporu iných protokolov. Využíva sa v Avaste na asynchrónnu komunikáciu medzi niektorými systémami. Tento oddiel primárne vychádza z informácií dostupných na zdroji [4].

Pre prácu s RabbitMQ sú podstatné nasledujúce koncepty:

- *Queue* — rad, do ktorého sa ukladajú prijímané správy a odkiaľ ich je možné čítať.

- *Exchange* — je to bod, na ktorý sa publikujú správy.

- *Binding* — naviazanie *queue* na *exchange*, ktoré zabezpečuje, že správy publikované na dané *exchange* sa dostávajú do tohto *queue*.

- *Routing key* — smerovací kľúč. Je parametrom pri vytváraní správy, ako aj pri vytváraní naviazania. Umožňuje presnejšie špecifikovať kam má byť správa zaslaná, respektíve ktoré správy majú byť prijímané.

- *Publisher* — aplikácia publikujúca správy na *exchange*.
- *Consumer* — aplikácia, ktorá spracúva správy prichádzajúce do *queue*. Väčšinou funguje tak, že má zadefinovanú funkciu, ktorá sa zavolá pri prijatí správy, pomocou ktorej ju spracuje.

Presné správanie pri distribuovaní správ na naviazané rady závisí od typu *exchange*. Základné typy sú nasledovné:

- *Direct exchange* — správy sú rozposlané všetkým radám, ktoré sú naviazané s rovnakým smerovacím kľúčom, ako má správa.
- *Fanout exchange* — správy sú rozposlané všetkým naviazaným radám, smerovací kľúč sa ignoruje.
- *Topic exchange* — smerovacie kľúče chápe ako hierarchicky zoradené témy, pričom sú úrovne hierarchie oddelené znakom bodky. Pri naväzovaní umožňuje smerovací kľúč špecifikovať pomocou vzorov, v ktorých môže byť na reprezentovanie ľubovolnej témy v jednej úrovni hierarchie použitý zástupný znak hviezdy, a na reprezentáciu ľubovolných tém v ľubovolnom počte hierarchií znak mriežky. Vďaka tomu dokáže robiť čokoľvek, čo je možné s typom *direct* aj s typom *fanout* a i niečo naviac. V Avaste je preferovaný z dôvodu jeho flexibility.

YARKA

YARKA je distribuovaná služba skenovania pre YARU. Je vyvíjaná spoločnosťou Avast. Využíva sa primárne za účelom testovania kvality pravidiel. YARKA umožňuje začínať skeny dvoma spôsobmi — buď ručne, pomocou rozhrania na webovej stránke, alebo cez webové API. Pri vytváraní žiadosti na sken je možné YARKE poslať ľubovolné pravidlá, avšak pri voľbe skenovaných súborov užívateľ iba vyberá zo sád, ktoré sú dostupných na YARKE. Väčšina sád obsahuje čisté súbory, ktoré sú určené na testovanie toho, či pravidlá použité na skenovanie vyvolávajú falošné pozitíva, ale existuje aj sada vzoriek vírusov. Tento oddiel čerpá z wiki stránky a súbora `README.md` interného Avastieho zdroja [12].

YARKA pozostáva z viacerých aplikácií, ktoré medzi sebou komunikujú. Konkrétne sú to tieto aplikácie:

- **Web** a **API** — časť zodpovedná za spracovanie požiadavkov od používateľov, taktiež sprístupňuje objekty databázy. Web umožňuje prácu s grafickým rozhraním.
- **Worker** — pracovník systému YARKA. Súčastne beží viacero inštancií, každá s unikátnym menom. Dostávajú úlohy, čo sú vlastne naplánované skeny, ktoré treba vykonať nad nejakou sadou pravidiel.
- **Master** — riadiaca časť YARKY. Distribuuje úlohy pracovníkom, výsledky skenov publikuje systémom RabbitMQ.

3.3 Životný cyklus pravidiel v Avaste

V tejto podkapitole je popísaný vznik, testovanie a vyhodnocovanie, ukladanie a aktualizácie YARA pravidiel v Avaste.

Ako bolo spomínané, v rámci Avastu sú YARA pravidlá používané viacerými systémami, do ktorých sa distribuujú automaticky z jednej centrálnej sady pravidiel. Tieto centralizované pravidlá sú uložené s využitím služby GitHub Enterprise v internom repozitári. Významnými vetvami repozitára sú hlavná vetva `master` a potom vetva `stable` obsahujúca iba tie pravidlá z `master`, ktoré úspešne prešli všetkými kontrolami a boli distribuované do daných systémov.

Do repozitára môžu byť pravidlá pridávané manuálne malvérovými analytikmi, prípadne automaticky niektorými systémami. Svoje pravidlá malvérový analytici zvyčajne tvoria na základe konkrétnej vzorky alebo vzoriek malvéru, buď manuálne alebo pomocou Avastieho nástroja na generovanie pravidiel **Yaragen**. Zmeny v hlavnej vetve repozitára sú obmedzené, práva vykonávať ich majú iba niekoľký analytici. Ostatný musia vytvoriť pull request s pridávanými alebo upravenými pravidlami a počkať na schválenie jednou z osôb s právom pridávania zmien do hlavnej vetvy.

Pred tým, než sa nejaké pravidlo začne využívať systémami Avastu, musí prejsť viacerými štádiami verifikácie a spracovania:

1. **Kompilácia** — kompilácia pravidiel aby sa detegovali syntaktické a sémantické chyby.
2. **Kontrola formátu** — kontrola či pravidlá dodržiavajú Avastové zásady pre písanie YARA pravidiel (ako boli popísané v podkapitole 3.1).
3. **Transformácie** — transformácie pravidiel do podôb vhodných pre rôzne rozličné systémy. Okrem transformácií pre konkrétny systém existujú aj všeobecnejšie transformácie, napr. transformácia všetkých pravidiel do jedného súboru.

Pre niektoré zo systémov nasleduje po transformácii ešte opätovná kontrola dodržovanie štandardov a prípadne ďalšie spracovanie. Na prechod pravidiel týmito týchto štádiami slúži Avastový nástroj **YARA rules toolchain**. Je spúšťaný automaticky pri pridávaní pravidiel do repozitára. Ak nastane chyba počas niektorej z týchto kontrol, zodpovedná osoba je musí napraviť.

Pre pravidlá, ktoré sú úspešne pridané do `master` vetvy repozitára pravidiel, **YARA rules hook** automaticky zabezpečí naplánovanie kontroly skenovacím nástrojom YARKA, čím sa otestuje, či vyvolávajú falošné pozitíva. Ak test zlyhá, YARA rules hook vygeneruje email s varovaním pre autorov commitov a ak pravidlá zodpovedné za zlyhanie nie sú kategórie `hunting`, repozitár navyše prechádza do `failed` stavu. V tomto stave sa pravidlá do systémov nedistribuujú, kým sa problém nevyrieši. Pre ušetrenie zdrojov prebiehajú skeny nástrojom YARKA pre skupiny commitov čakajúcich na kontrolu naraz, pri čom chyba spôsobuje notifikáciu autorov všetkých commitov a konkrétne problematické pravidlá sú identifikované pomocou heš hodnoty commitu. Commity, ktoré úspešne prejdú testami, sú automaticky distribuované do systémov Avastu a pridané do chránenej vetvy `stable`.

Kapitola 4

Návrh systému pre zber, vyhodnotenie, a aktualizáciu YARA pravidiel

V tejto kapitole je popísaný cieľ systému, identifikované externé zdroje, návrh a funkcionalita systému ako celku a následne aj jednotlivých častí systému.

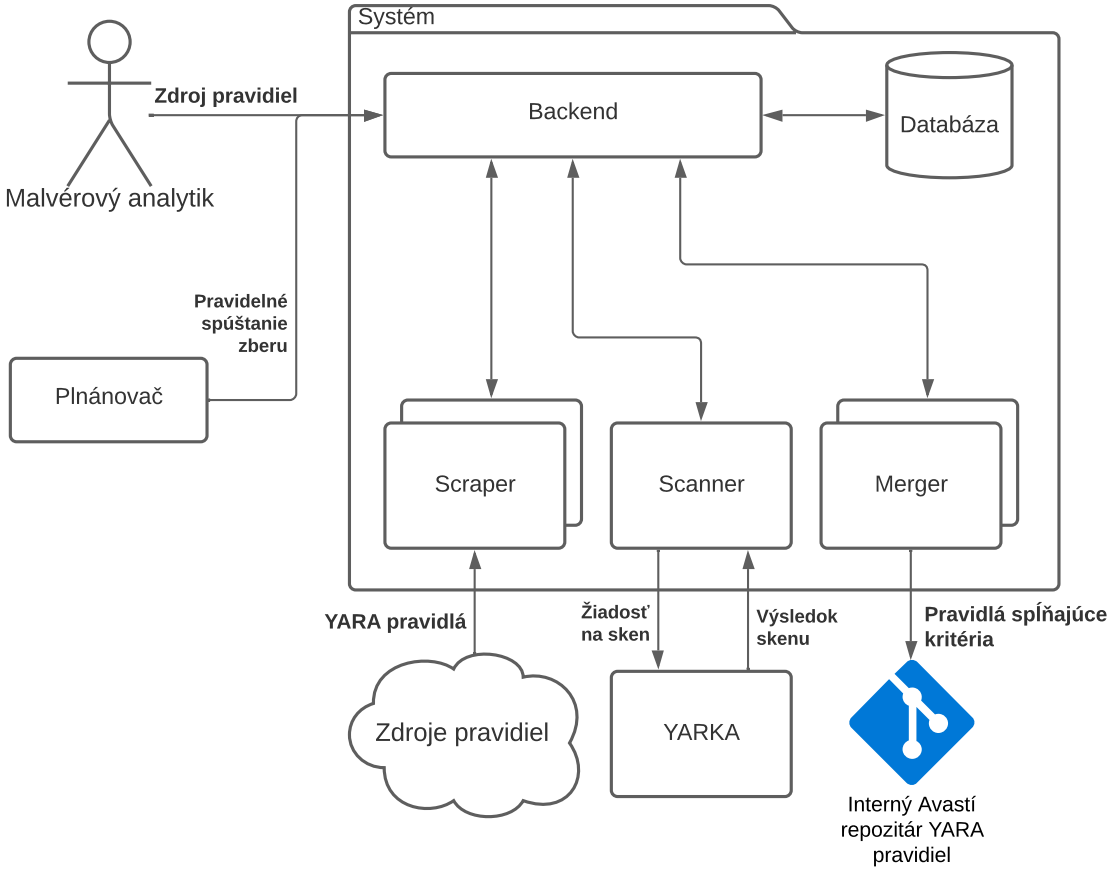
Cieľové správanie systému je nasledovné. Systém na vstupe dostane adresu zdroja voľne dostupných YARA pravidiel. Daný zdroj zaradí medzi sledované zdroje. Systém bude taktiež umožňovať manažment daných zdrojov, a jednotlivých pravidiel získaných zo zdrojov, ako napríklad aktiváciu alebo deaktiváciu zvoleného zdroja, pričom deaktivované zdroje budú ignorované. Zo všetkých sledovaných zdrojov budú pravidelne získavané pravidlá. Pri aktualizácii zdroja budú v rámci možností identifikované vzťahy medzi starými a novými pravidlami. Ďalej budú získané pravidlá testované skenovaním vzoriek súborov Avastu, podľa čoho bude vyhodnotená kvalita pravidla. Pravidlá, ktoré budú spĺňať požadované nároky budú upravené do podoby vhodnej pre začlenenie do Avastieho úložiska pre YARA pravidlá, a následne doň budú aj začlenené.

Ako vhodné externé zdroje pre účely využitia systémom tejto práce boli identifikované tieto zdroje:

- **Yara Rules Project**[1] — verejne dostupný Git repozitár obsahujúci rozsiahle množstvo pravidiel roztriedených podľa typu. Tento projekt je určený na pokrytie potreby malvérových analytikov mať jeden repozitár kde sú YARA pravidlá zbierané, klasifikované a udržované aktuálne. Má slúžiť ako otvorená komunita [14].
- **Malpedia**[2] — pravidelne aktualizovaný zdroj ponúkajúci vzorky malvéru, ako aj súvisiace YARA pravidlá. Získať sa je dá pomocou API alebo z Git repozitáru. Malpedia obsahuje aj dôverné informácie, poskytované iba členom, ktoré nesmú byť zverejňované. Na označenie toho, či je konkrétne pravidlo dostupné verejne alebo utajované, sa používa TLP — traffic light protocol (protokol semaforu) — pričom verejne dostupné pravidlá sú týmto protokolom označené ako `WHITE` [9].
- **Reversing Labs**[3] — pravidlá vytvorené malvérovými analytikmi firmy ReversingLabs, sprístupnené vo forme Git repozitáru. Obsahuje detečné pravidlá (nie „lovecké“

[1] https://github.com/Yara-Rules/rules
[2] https://malpedia.caad.fkie.fraunhofer.de/
[3] https://github.com/reversinglabs/reversinglabs-yara-rules

pravidlá), čiže cieľom je byť čo najprecíznejší bez straty kvality a nevyvolávať žiadne falošné pozitíva.



Obr. 4.1: Diagram systému ako celku

Systém sa bude skladať z rozhrania systému zodpovedného za prezentáciu informácií o systéme a za jeho riadenie pod názvom *backend*, z databázy na ukladanie všetkých potrebných dát a z troch komponentov poskytujúcich jednotlivé služby — *scraper* zodpovedný za zber pravidiel zo zdroja, *scanner* zodpovedný za testovacie skeny so zozbieranými pravidlami a *merger* zodpovedný za začlenenie vhodných pravidiel do úložiska pravidiel Avastu. Ďalej bude systém využívať plánovač úloh na pravidelné spúšťanie zberu pravidiel, po dokončení ktorého naň automaticky naviaže sken a následne zber. Diagram systému možno vidieť na obrázku 4.1.

4.1 Backend

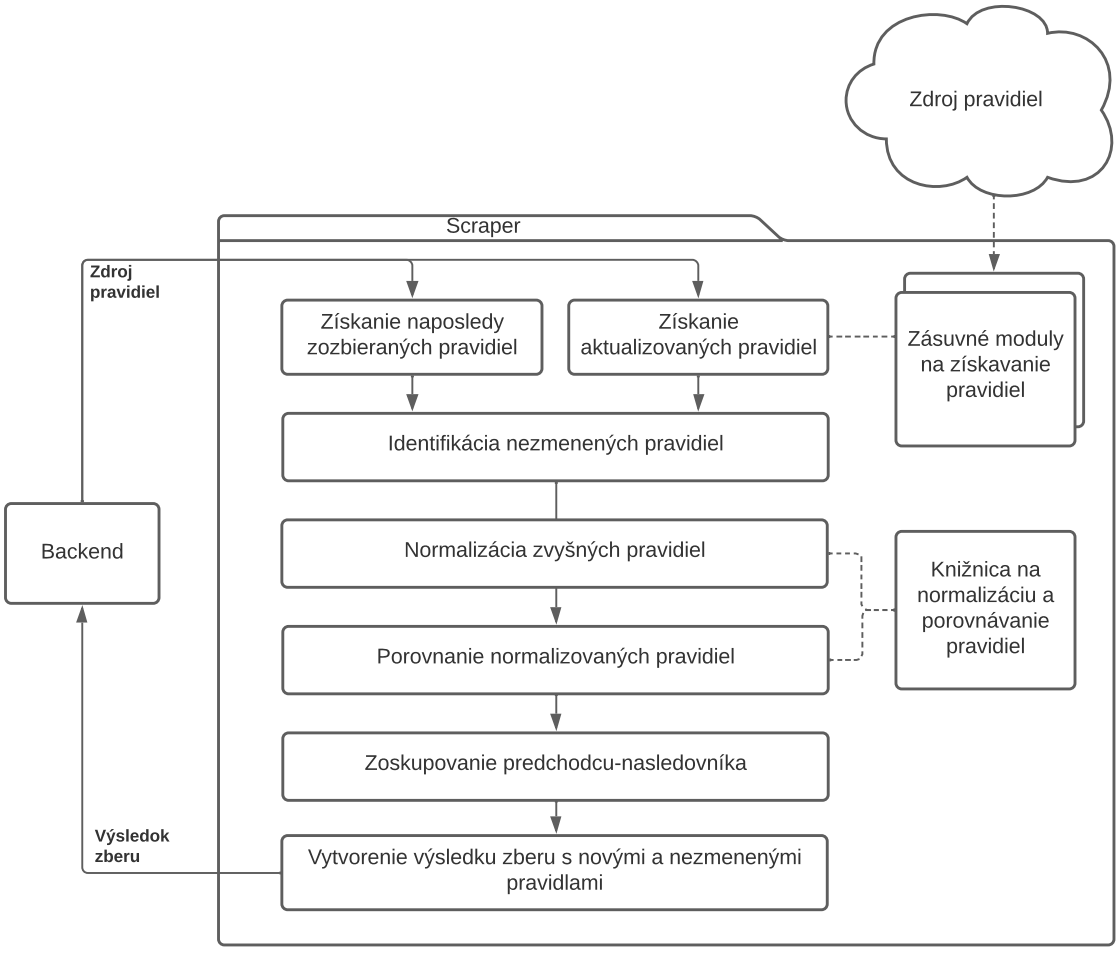
Backend bude umožňovať užívateľom pracovať so systémom. Umožní užívateľom čítať alebo pridávať zdroje, ktoré sú uložené v databáze a taktiež sprístupní zozbierané pravidlá. Ďalej umožní spúšťanie jednotlivých služieb a prijímanie výsledkov z činností týchto služieb.

Žiadosti na zber bude backend posielať jednotlivo, každá správa bude spúšťať jeden zber pre jeden zdroj. Toto umožní distribúciu zberov v prípade behu viacerých inštancií služby

scraper, vďaka čomu bude možné službu jednoducho škálovať. Rovnako sa bude postupovať pri žiadostiach na merger. Pri službe scanner sa budú všetky pravidlá vyžadujúce sken posielať súčasne. Nemá zmysel uvažovať viacero inštancií tejto služby alebo rozdelenie žiadostí na sken na menšie celky, keďže spúšťať viacero menších skenov je menej efektívne, a teda cieľom je vždy spustiť jeden spoločný sken pre všetky relevantné pravidlá.

4.2 Scraper

Kompletný zoznam úloh služby scraper je získavanie pravidiel či ich aktualizácií zo zdroja a následné vyhodnocovanie vzťahov medzi starými a novými pravidlami (vzťah predchodca-nasledovník). Na vstupe dostane požiadavok na získanie pravidiel pre zvolený zdroj, výstupom bude zoznam nezmenených a zoznam novo získaných pravidiel. Súčasťou komponenty budú zásuvné moduly pre rôzne typy zdrojov a knižnica, umožňujúca pravidla normalizovať (čiže sémanticky ekvivalentné ale syntakticky rozdielne výrazy jazyka previesť do jednotnej podoby) a porovnávať. Diagram práce tejto časti možno vidieť na obrázku 4.2.



Obr. 4.2: Diagram práce komponenty scraper

Základný tok práce tejto komponenty je nasledovný. Scraper dostane z časti backend záznam zdroja, spolu s naposledy zozbieranými pravidlami pre tento zdroj. Podľa adresy

zdroja budú s využitím vhodného zásuvného modulu pre daný typ zdroja získané aktualizované pravidlá. Najprv sa medzi už neaktuálnymi a novými pravidlami nájdu kompletne identické pravidlá. Pravidlá, pre ktoré nebolo nájdené identické pravidlo, budú normalizované, a následne budú neaktuálne a nové pravidlá medzi sebou porovnávané. Na základe podobnosti rôznych častí bude rozhodnuté, či sa jedná o to isté, aktualizované pravidlo alebo dva rozdielne pravidla. Novým pravidlám, ktoré boli identifikované ako nasledovníci neaktuálnych pravidiel, bude priradený ich predchodca. Pre prvý zber z daného zdroja budú kroky súvisiace s porovnávaním pravidiel predchádzajúceho zberu vynechané. Nakoniec sa pošle výsledok zberu s nezmenenými a novými pravidlami časti backend.

Pre túto prácu je primárne uvažovaný jeden zásuvný modul na získavanie pravidiel, pre úložiská využívajúce systém Git. Bude ale umožnené vytvárať nové zásuvné moduly, ak implementujú požadované rozhranie.

Normalizácia

Normalizácia YARA pravidiel je v tomto kontexte uvažovaná ako kombinácia viacerých čiastkových normalizácií. Keďže sa tieto čiastkové normalizácie budú navzájom ovplyvňovať, v praxi je potrebné vykonávať ich v správnom poradí.

Jednou z najpodstatnejších čiastkových normalizácií je normalizácia logických výrazov. Ide o normalizáciu vzhľadom k používaniu operátorov `and`, `or` a `not` a literálom `true` a `false`. Princíp spočíva v prevedení logických výrazov do konjunktívnej normálnej formy. Konjunktívna normálna forma je forma logického výrazu, kedy je výraz vyjadrený ako konjunkcia disjunkcií premenných. Ako premenná sa v tomto kontexte chápe akýkoľvek výraz iný, ako operácie spomínaných logický operátorov, logickej hodnoty `true` a logickej hodnoty `false`. Pre túto normalizáciu je tiež potrebné zvoliť kritérium jednoznačného zoradenia rovnocenných prvkov výrazov (napr. zoradenie podľa textu reprezentujúceho jednotlivé prvky). Táto normalizácia by sa avšak mala vykonávať ako posledná, keďže ostatné čiastkové normalizácie by mohli generovať alebo upravovať logické výrazy, prípadne zmeniť výrazy tak, že by to ovplyvnilo ich zoradenie.

Ďalším typom čiastkovým normalizácií sú normalizácie zamerané na zbavenie sa odlišností spôsobených rôznymi názvami zadefinovaných reťazcov s rovnakými hodnotami. Takéto normalizácie sú konkrétne dve. Prvá nahradí potrebné všetky identifikátory reťazcov so zástupným znakom zoznamom reťazcov, ktorým tento identifikátor odpovedá. Druhá zmení identifikátory reťazcov tak, aby priamo záviseli iba na význame reťazca. To znamená, že pre každú konkrétnu kombináciu nejakej hodnoty a modifikátorov reťazca bude jednoznačne definovaný konkrétny názov a naopak, pričom tento vzťah bude 1:1.

Podobná normalizácia je potrebná aj pre identifikátory pravidiel, ktoré sa nachádzajú v podmienke. Tie musia byť nahradené podmienkou pravidla, ktoré má tento identifikátorom. Pre to, aby výsledné pravidlo bolo legitímne pravidlo v jazyku YARA, musí byť súčasťou tejto zmeny aj začlenenie reťazcov identifikovaného pravidla medzi reťazce normalizovaného pravidla. Toto musí byť vykonané až po normalizácií reťazcov, ktorá bola opísaná v predošlom odseku, aby bolo zaistené, že nenastane kolízia dvoch reťazcov s rovnakými identifikátormi ale odlišnými hodnotami.

Posledným typom normalizácií je úprava špecifických konštrukcií jazyka YARA, ktoré majú rovnaký význam, na zvolenú jednotnú formu:

- Nahradenie výrazu `them` výrazom `($*)`. Toto treba vykonať pred prvou zo spomínaných normalizácií reťazcov, keďže tá pracuje aj s výrazmi `($*)`.

- Zjednodušenie reťazcových cyklov s podmienkou `($)` na `<amount> of <iterable>` výraz.
- Prevod `all of <iterable>` a `any of <iterable>` výrazov na ekvivalentnú konjunkciu alebo disjunkciu.
- Pridanie explicitných indexov tam, kde sú použité implicitné.

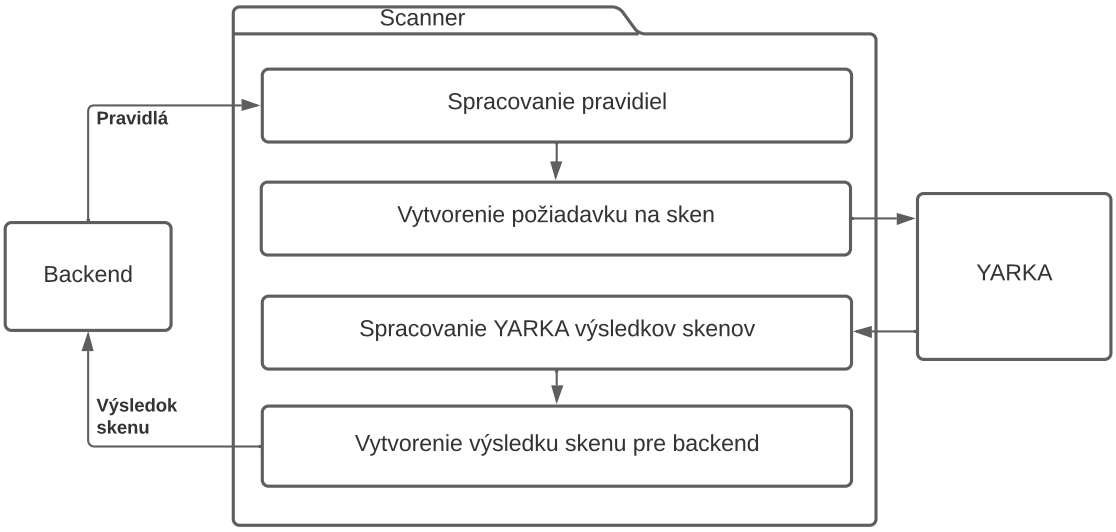
Porovnávanie pravidiel

Pravidlá sa budú porovnávať po jednotlivých častiach, tak ako boli spomenuté v podkapitole 2.2. Pri porovnávaní budú zanedbané biele znaky a komentáre, ako aj poradie jednotlivých prvkov v zoznamoch (napr. v zozname meta informácií). Výsledkom bude informácia o zhode medzi jednotlivými časťami. Ako sémantickú zhodu rozumieme zhodu (normalizovaných) podmienok.

Vyhodnocovanie porovnávania

Zhody medzi dvojicami pravidiel budú ohodnotené podľa kvality zhody. Ako minimálne kritérium pre zhodu bola zvolená sémanticky identická podmienka, následne má na kvalitu vplyv aj názov pravidla a cesta v rámci zdroja, a nakoniec je relevantná aj zhoda ostatných častí. Určovanie dvojíc predchodca-nasledovník bude vychádzať z kvality zhody, čiže pre vzájomne vylučujúce sa párovania bude zvolené to, ktoré má vyššiu kvalitu zhody. Po priradení zistených predchodcov novým pravidlám je možné tieto pravidlá, spolu s nezmenenými pravidlami, poslať v rámci výsledku zberu pre daný zdroj časti backend.

4.3 Scanner



Obr. 4.3: Diagram práce komponenty scanner

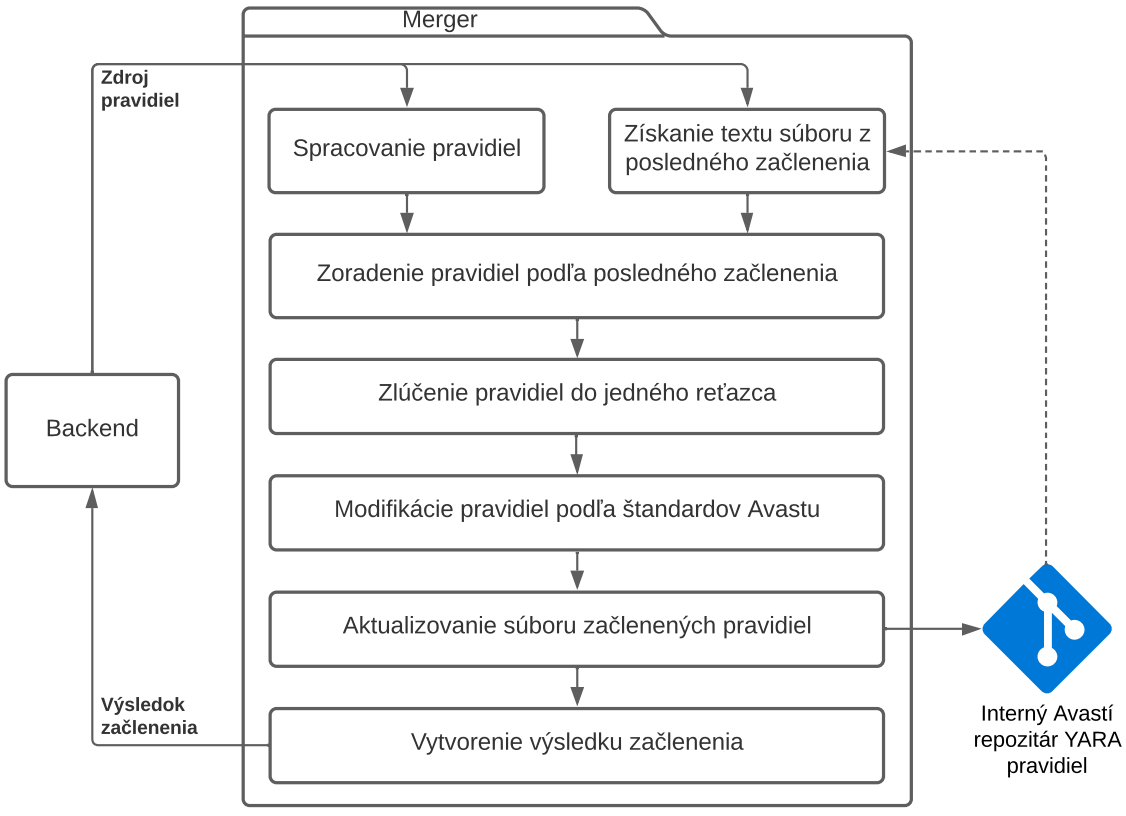
Táto komponenta má za úlohu zabezpečiť spúšťanie testovacích skenov so zozbieranými pravidlami pravidlami na overenie ich kvality, prevzatie a spracovanie výsledkov skenov, ich

vyhodnotenie a nakoniec zaslania výsledku časti backend. Na samotné skenovanie využíva Avastí nástroj YARKA (pre viac informácií pozri podkapitolu 3.2). Diagram práce tejto časti možno vidieť na obrázku 4.3.

Služba na vstupe dostane zoznam pravidiel obsahujúci pravidlá, pre ktoré je nutné vykonať sken. Pravidlá musí spracovať do vhodnej podoby, aby boli zlúčené do jednoho reťazca, pričom treba dbať na vyhnutie sa kolíziám mien, správne používanie modifikátora `private` (sledované pravidlá ho nesmú mať) a `global` (v zoskupenom reťazci by mal nežiadúci účinok). S týmto reťazcom pravidiel vytvorí žiadosť na službu YARKA, aby boli pravidlami oskenované sady čistých súborov. Po ukončení skenovania modul prijme od služby YARKA správu o dokončení. Scanner výsledky spracuje, prípadne požiada YARKU o dodatočné informácie a na základe nich vytvorí výsledok skenovania pre časť backend.

4.4 Merger

Táto komponenta má spravovať pridávanie pravidiel do interného úložiska Avastu. Musí zaisťovať vhodnú formu pravidiel aj v možnej miere zabezpečiť toleranciu a zachovanie interných modifikácií, ktoré na tieto pravidlá boli aplikované v rámci Avastu. Pravidlá budú začleňované do súborov tak, že jeden súbor bude vždy obsahovať všetky pravidlá jedeného zdroja. Diagram práce tejto časti možno vidieť na obrázku 4.4.



Obr. 4.4: Diagram práce komponenty merger

Proces začleňovania začína prijatím správy obsahujúcou zdroj s jeho pravidlami. Pre daný zdroj merger navyše získa z repozitára Avastu súbor s jeho aktuálne začlenenými

pravidlami. Z pravidiel zdroja prijatých v správe sa vyberú tie, ktoré spĺňajú kritériá pre pridanie do repozitára, čiže sú aktuálne a pri testovacom skenovaní nevyvolávali falošné pozitíva. Následne sa pravidlá spracujú podobným spôsobom ako pravidlá spracované scannerom v predošlej podkapitole, aby bolo možné všetky pravidlá pre daný zdroj zlúčiť do jedného reťazca či súboru. Pri zlučovaní aktuálne spracovaných pravidiel do jedného reťazca sa tieto pravidla zoradia podľa ich pozícií alebo pozícií ich predchodcov v získanom súbore, čím sa minimalizujú zmeny. Nové pravidlá sa zaradia na koniec. Na takto zlúčené pravidlá sa potom ešte aplikujú modifikácie (formátovanie, pridanie meta informácií, ...), aby spĺňali štandardy Avastu. Teraz je už možné obsah súboru pre daný zdroj na Gite aktualizovať novo vygenerovaným obsahom a vytvoriť pull request. Nakoniec sa na backend môže poslať výsledok začlenenia označujúci, ktoré pravidlá sú aktuálne začlenené.

Konkrétna stratégia aplikovania zmien na Gite je nasledovná. Pre každý zdroj sa bude udržovať jedna vetva, ktorá bude vytvorená pri prvom začleňovaní. Každé začlenenie pre konkrétny zdroj do jeho vetvy pridá jeden commit aktualizujúci súbor zdroja. Pre každý jeden commit bude taktiež vytvorená nová vetva, bez žiadnych prídavných zmien. Táto vetva bude vytvorená čisto za účelom otvorenia pull requestu z tejto vetvy do vetvy `master`. V prípade konfliktu, spôsobeným internými modifikáciami pravidla, ktorý Git nebude schopný vyriešiť automaticky, sa konflikt môže vyriešiť manuálne v tejto vetve s využitím grafického užívateľského rozhrania GitHubu (alebo štandardne cez Git). Týmto sa vyhne zasahovaniu do vetvy, s ktorou priamo pracuje merger, a zároveň akýkoľvek manuálne vyriešený konflikt bude Gitom automaticky braný do úvahy aj pri ďalších pull requestoch.

Kapitola 5

Implementácia systému pre zber, vyhodnotenie, a aktualizáciu YARA pravidiel

Táto kapitola obsahuje informácie o implementácií systému, ktorý je predmetom tejto práce. Najprv sú uvedené všeobecné informácie o systéme. Následne je popísaný kód, ktorý je zdieľaný viacerými časťami. Ďalšia podkapitola sa venuje riadiacej časti pod názvom backend. Posledné podkapitoly sa venujú jednotlivým službám systému, čiže častiam scraper, scanner a merger.

V rámci implementácie je systém ako celok označovaný názvom `scraper` (názov zdieľaný s jednou z čiastkových častí). Samotný zdrojový kód systému je rozdelený na 5 priečinkov — jeden priečinok pre každú z už spomínaných štyroch častí, plus piaty priečinok `common` obsahujúci kód, ktorý je zdielaný viacerými časťami. Súčasťou tejto práce je aj knižnica na normalizáciu a porovnávanie pravidiel s názvom `spire`. Táto knižnica je využívaná časťou scraper a jej implementácia je oddelená od samotného systému.

Systém vytvorený tak, že každá z jednotlivých častí beží v izolovanom prostredí, tzv. kontajneri, ktoré medzi sebou môžu komunikovať. Toto je zaistené s využitím softvéru Docker[1]. Okrem základných štyroch aplikačných častí systém ešte využíva relačnú SQL databázu s dialektom postgres[2] a *broker* RabbitMQ[3] (pre bližšie informácie pozri podkapitolu 3.2).

Ako implementačný jazyk bol zvolený Python 3 (verzia 3.8<=), pričom sa využíva jeho asynchrónna funkcionalita. Vybrané Python knižnice, ktoré sú používané naprieč viacerými časťami systému, sú:

- `yaramod`[4] — knižnica vyvinutá spoločnosťou Avast. Umožňuje spracovať YARA pravidlá zo súboru alebo reťazca do Python objektu. Čiastočne aj umožňuje tento objekt upravovať, a to konkrétne jeho meno, meta informácie a podmienku. Podmienka je reprezentovaná abstraktným syntaktickým stromom (AST). S podmienkami sa pracuje tak, že sa vytvorí trieda dediaca z `yaramod` triedy `ObservingVisitor` alebo `ModifyingVisitor` (prvý slúži na čítanie, druhý aj úpravu). Spracovanie podmienky sa spúšťa metódou `observe` alebo `modify`. Táto metóda postupne prechádza AST vo-

[1] https://www.docker.com/
[2] https://www.postgresql.org/
[3] https://www.rabbitmq.com/
[4] https://yaramod.readthedocs.io

laním preddefinovaných metód `visit_<typ-výrazu>Expression`. Užívateľ tieto metódy predefinuje podľa potreby, čím sa zaistí požadovaná funkcionalita [8].

- `requests`[5] — knižnica na odosielanie HTTP požiadavkov a spracovanie ich odpovedí.
- `aio_pika`[6] — knižnica na asynchrónnu prácu s RabbitMQ.
- `aiomisc`[7] — z tejto knižnice je použitý dekorátor funkcií `@threaded`, ktorý zabezpečí, že pri volaní sa tieto funkcie spustia v novom vlákne. V implementácií systému bol použitý na časovo náročné funkcie jednotlivých služieb, aby aj počas ich vykonávania mohli v pozadí naďalej pravidelne prebiehať akcie potrebné na udržovanie aktívneho RabbitMQ pripojenia.

5.1 Kód zdielaný viacerými časťami

Ku kódu, ktorý je v zdieľanom priečinku `common` patrí spracovanie konfigurácie, zadefinovanie schém používaných na komunikáciu zo systémom či v rámci systému a tiež všeobecné utility pre prácu z objektami reprezentujúcimi pravidlá.

Konfigurácia

Konfigurácia systému je uložená v súbore `config.conf`. Formát tohto súboru je HOCON[8]. Pomocou knižnice `pyhocon`[9] je tento súbor spracovaný a na základe hodnôt v tomto súbore sú nastavené premenné jazyku Python, ktoré sa následne môžu importovať do iných modulov.

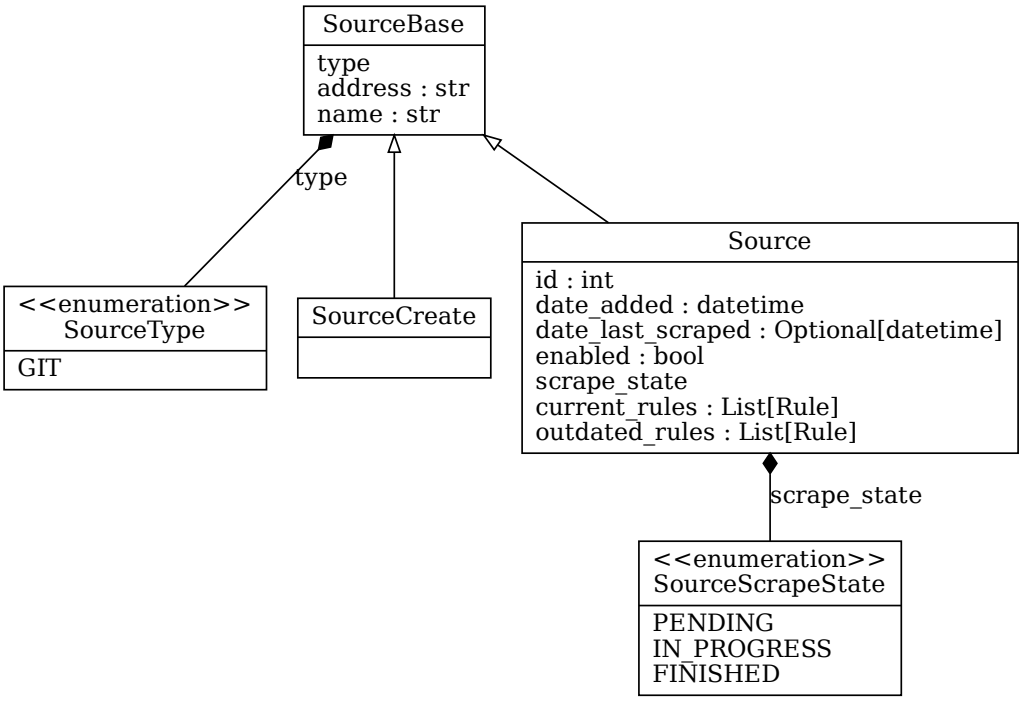
Konfigurácia sa týka nasledujúcich aspektov aplikácie:

- Databáza — aká je adresa, prihlasovacie údaje, ...
- Logovanie — aký je stupeň logovania (`DEBUG`, `INFO`, ...).
- Backend — na akej adrese je pre scraper, scanner a merger dostupný backend.
- RabbitMQ — používaný *exchange*, čas expirácie správ, ...
- YARKA — na akej adrese je dostupná YARKA, aké sady súborov sú využívané, ...
- GitHub — aký repozitár je používaný, ktoré vetvy, ...
- Meta informácie — aké hodnoty meta informácií vyžadovaných Avastom sú pravidlám nastavené.

[5] https://docs.python-requests.org/en/latest/
[6] https://aio-pika.readthedocs.io/en/latest/index.html
[7] https://pypi.org/project/aiomisc/
[8] https://github.com/lightbend/config/blob/main/HOCON.md
[9] https://github.com/chimpler/pyhocon

Schémy

Na definíciu tried schém určených na komunikáciu bola zvolená knižnica `pydantic`[10]. Táto knižnica zabezpečuje konverziu z/na formát JSON či pythonový `dict` (asociatívne pole), alebo z databázového modelu, pri čom taktiež vykonáva validáciu oných dát. Schémy je možné vidieť na obrázkoch 5.1, 5.2 a 5.3. Pre zdroje aj pravidlá existujú 2 verzie schém. Jedna verzia je používaná pri pridávaní nového pravidla alebo zdroja do databázy. Druhá verzia reprezentuje existujúce pravidlo či zdroj, so všetkými dátami, ktoré sú vytvorené až pri vkladaní do databázy (napr. ID alebo dátum pridania). Obe verzie pre daný typ objektu dedia zo spoločnej triedy, ktorá obsahuje zdielané atribúty. Zdroje aj pravidlá je možné deaktivovať (a zase aktivovať), kedy budú ignorované pri skenovaní a začleňovaní.



Obr. 5.1: Schémy zdroja

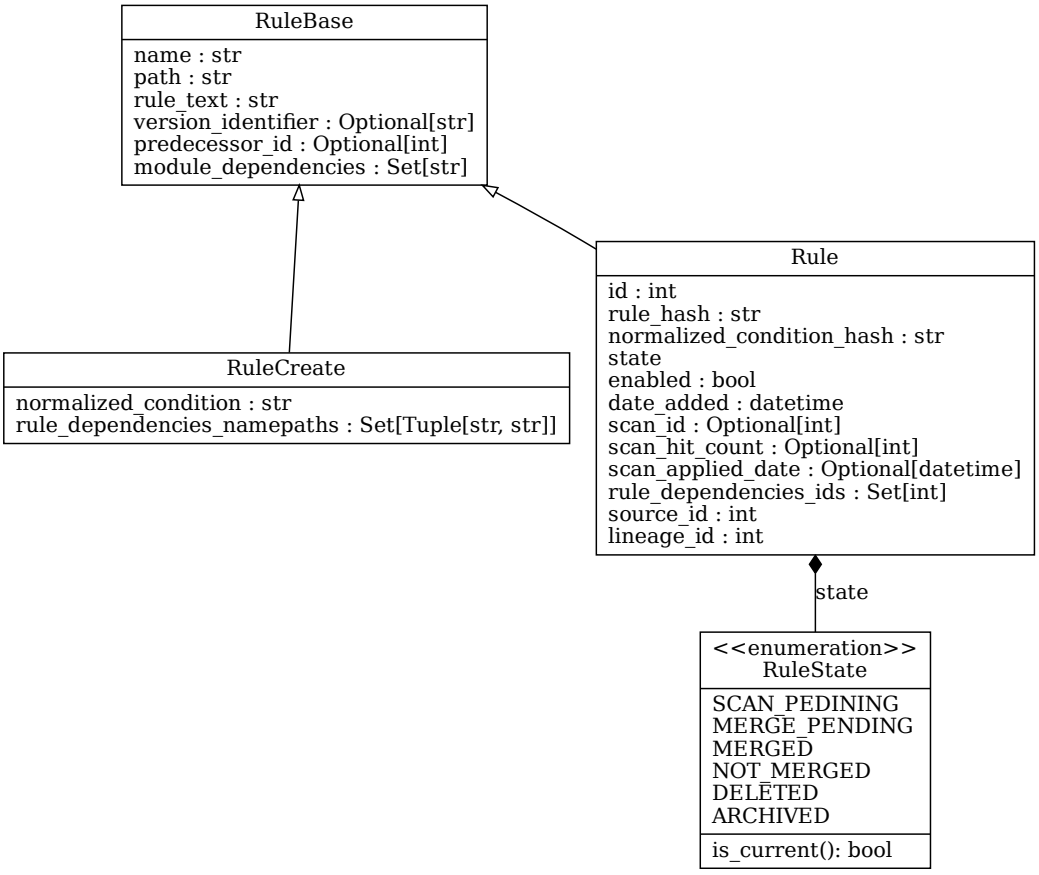
Obe schémy zdroja, dediace z triedy `SourceBase`, obsahujú tieto atribúty:

- `type` — typ zdroja. Aktuálne je podporovaný jeden typ — `GIT`, pre zdroje využívajúce systém Git.
- `address` — adresa zdroja. Presná syntax a sémantika závisí od typu zdroja.
- `name` — alfanumerické meno priradené zdroju užívateľom, ktorý ho pridal do tohto systému. Musí byť v rámci databázy unikátne.

Schéma na pridanie zdroja `SourceCreate` neobsahuje žiadne dodatočné atribúty. Schéma už uloženého zdroja `Source` použitá pri čítaní navyše obsahuje:

[10] https://pydantic-docs.helpmanual.io/

- `id` — ID zdroja, ktoré mu bolo pridelené databázou, a je v rámci všetkých zdrojov v databáze unikátne.
- `date_added` — časový údaj pridania zdroja do databázy.
- `date_last_scraped` — časový údaj aplikovania posledného výsledku zberu.
- `enabled` — stav aktivácie zdroja.
- `scrape_state` — aktuálny stav zberu. Môže byť: očakávaný zber, prebiehajúci zber, ukončený zber.
- `current_rules` — zoznam aktuálnych zozbieraných pravidiel pre daný zdroj.
- `outdated_rules` — zoznam neaktuálnych zozbieraných pravidiel pre daný zdroj.



Obr. 5.2: Schémy pravidla

Obe schémy pravidla, dediace z triedy `RuleBase` obsahujú tieto atribúty:

- `name` — meno pravidla.
- `path` — cesta k pravidlu na zdroji.

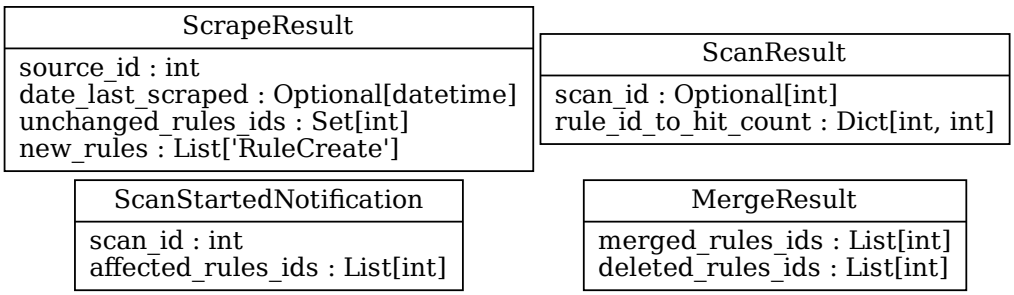
- `rule_text` — text pravidla.
- `version_identifier` — identifikátor verzie. Pre pravidlá zo zdroja typu `GIT` je to heš commitu z ktorého bolo pravidlo získané.
- `predecessor_id` — ID predchodcu.
- `module_dependencies` — moduly vyžadované pravidlom.

Schéma pravidla `RuleCreate`, slúžiaca na vytvorenie pravidla, navyše obsahuje:

- `normalized_condition` — normalizovaná podmienka pravidla používaná na získanie hešu normalizovanej podmienky.
- `rule_dependencies_namepaths` — zoznam dvojíc mena a cesty pravidiel na ktorých je toto pravidlo závislé. Takáto dvojica unikátne identifikuje pravidlo v rámci sady súčasne aktuálnych pravidiel zvoleného zdroja, keďže v jednom súbore nemohli byť súčasne dva rovnako pomenované pravidlá.

Schéma už uloženého pravidla `Rule` použitá pri čítaní navyše obsahuje:

- `id` — ID pravidla, ktoré mu bolo pridelené databázou, a je v rámci všetkých pravidiel v databáze unikátne.
- `rule_hash` — SHA256[11] heš textu pravidla, umožňujúci jednoduchšiu detekciu identických pravidiel.
- `normalized_condition_hash` — SHA256 heš normalizovanej podmienky pravidla, umožňujúci jednoduchšiu detekciu sémanticky ekvivalentných pravidiel.
- `state` — stav pravidla. Existuje 6 možných stavov: očakáva sa sken, očakáva sa začlenenie do repozitára Avastu, nezačlenené do repozitára Avastu, začlené do repozitára Avastu, vymazané, archivované. Neaktuálne pravidlá sú označené ako archivované. Výnimkou sú pravidlá, ktoré boli začlenené do repozitára Avastu, a následne sa stali neaktuálnymi. Takéto pravidlá sú označené ako vymazané zo zdroja, pretože ich musí spracovať merger odstránením z úložiska Avastu, po čom už môžu bezpečne prejsť do stavu archivovaných pravidiel.
- `enabled` — stav aktivácie pravidla.
- `date_added` — časový údaj pridania pravidla do databázy systému..
- `scan_id` — ID YARKA skenu zodpovedného za toto pravidlo.
- `scan_hit_count` — počet zistených zhôd pri YARKA skene.
- `scan_applied_date` — časový údaj pridania výsledkov skenu.
- `rule_dependencies_ids` — ID pravidiel, na ktorých je toto pravidlo závislé.
- `source_id` — ID zdroja ku ktorému toto pravidlo patrí.
- `lineage_id` — ID „rodokmeňa“ pravidla. Toto ID unikátne identifikuje skupinu pravidiel tvoriacu líniu predchodcov/potomkov. Je rovné hodnote ID najstaršieho pravidla danej skupiny.



Obr. 5.3: Schémy pre zber, skenovanie a začleňovanie

Okrem schém zdrojov a pravidiel sú tu aj schémy na účelovú komunikáciu medzi službami a časťou backend. Tieto schémy sú konkrétne: schéma výsledku zberu, schéma notifikácie o začatí skenu, schéma výsledku skenu, schéma výsledku začlenenia.

Schéma výsledku zberu `ScrapeResult` obsahuje:

- `source_id` — ID zdroja, pre ktorý bol zber vykonaný.
- `date_last_scraped` — časový údaj posledného zberu pre daný zdroj. Zabezpečuje korektné chovanie v prípade, že by sa pre jeden zdroj spustil nový zber skôr, ako predošlý skončil. V takom prípade by sa po aplikovaní skôr dokončeného zberu tento údaj pre zdroj zmenil a druhý výsledok zberu by bol odmietnutý z dôvodu neaktuálnosti tohto údaju.
- `unchanged_rules_ids` — zoznam ID pravidiel, ktoré sa od posledného zberu nezmenili.
- `new_rules` — zoznam objektov reprezentujúcich nové pravidlá (vrátanie pravidiel, ktoré sú nasledovníkmi predošlých pravidiel)

Schéma notifikácie o započatí skenu `ScanStartedNotification` obsahuje:

- `scan_id` — ID spusteného YARKA skenu. V prípade, že sken nebol začatý z toho dôvodu, že nula pravidiel potrebuje sken, toto ID nie je nastavené na `None`.
- `affected_rules_ids` — zoznam ID pravidiel, pre ktoré bol sken spustený.

Schéma výsledku skenu `ScanResult` obsahuje:

- `scan_id` — ID YARKA skenu, ktorého sa tento výsledok týka.
- `rule_id_to_hit_count` — asociatívne pole priraďujúce ID pravidiel, ktoré vyvolali aspoň jednu zhodu pri skenovaní, k počtu týchto zhôd.

Schéma výsledku začlenenia `MergeResult` obsahuje:

- `merged_rules_ids` — zoznam ID pravidiel, ktoré boli začlenené.
- `deleted_rules_ids` — zoznam ID pravidiel, ktoré boli spracované ako vymazané.

[11] https://datatracker.ietf.org/doc/html/rfc6234

Utility na prácu s pravidlami

Pri práci s viacerými pravidlami dáva zmysel uvažovať abstrakciu sady pravidiel. Ide o zoradený zoznam pravidiel (tak aby sa prípadné závislosti vyskytli skôr ako pravidlá na nich závislé) a sadu modulov týmito pravidlami vyžadovanými. Za účelom reprezentácie tejto abstrakcie bola vytvorená trieda `Ruleset`. Interne je reprezentovaná zoradenou asociatívnym poľom pravidiel `OrderedDict` (kde pythonové id objektu je použité ako kľúč zabezpečujúci neopakovaný výskyt pravidla) a sadou modulov typu `set`. Pre túto triedu bola implementovaná funkcia prevodu na reťazec, čo umožňuje jednoducho získať výpis akejkoľvek sady pravidiel. Poradie modulov pri výpise bolo zadefinované deterministicky (abecedne) pre vyššiu konzistenciu.

Na tvorbu objektov `Ruleset` sa používajú objekty triedy `RulesetBuilder`. Pri inicializácií je možné zadefinovať ako pre objekty pravidla pristupovať k ich závislostiam alebo využívaným modulom, čo zabezpečuje generickosť pre rôzne objektové reprezentácie pravidiel. Následne pomocou metódy `get_full_ruleset` je možné pre ľubovolnú sadu pravidiel získať kompletnú sadu obsahujúce všetky závislosti, vrátane tých zdedených, čo sa zabezpečuje rekurzívnym volaním. Boli využité princípy dynamického programovania, a teda objekt obsahuje internú keš. Tá pre jednotlivé pravidlá uchováva v minulosti získané kompletné sady, čo zabraňuje opakovaniu identických volaní.

Menej generická je trieda `SchemaRulesPreprocessor` umožňujúca jednoduché spracovanie pravidiel reprezentovaných objektom schémy zo spoločnej časti. Je určená pre spracovanie pravidiel do takej formy, ktorá je vhodná pri snahe skombinovať veľkú sadu pravidiel do jedného súvislého textu. Objekty tejto triedy sa inicializujú zoznamom ID pravidiel, o ktoré je záujem pri výpise, a zoznamom objektov pravidiel, ktorý musí identifikované pravidlá a ich závislosti obsahovať. Navyše si takýto objekt aj vytvorí vlastnú inštanciu `RulesetBuilder`, inicializovanú tak, že je schopná pracovať so závislosťami, napriek tomu, že sa odkazujú na ID závislostí a nie samotné závislostí. Použitá je táto lambda funkcia:

```
lambda rule: [self.rule_by_id[id] for id in rule.rule_dependencies_ids]
```

Táto lambda funkcia k sebe naviaže asociatívne pole priraďujúce objekt reprezentujúci pravidlo k ID tohoto pravidla, čo umožňuje získavať zoznam objektov pravidiel miesto zoznamu ich ID. Následne poskytuje tieto funkcie:

- `rename_rules_to_ids` — pravidlá budú premenované na ich ID, s prefixom „_“, čo zabráni kolízií mien a umožní jednoducho spätne identifikovať dané pravidlo. Keďže je potrebné aj správne zmeniť identifikátory pravidiel, ktoré boli použité ako referencia v rámci podmienky iného pravidla, nie je možné ako riešenie využiť jednoduchú substitúciu. Miesto toho sa spracúvajú pravidlá po skupinách s rovnakým zdrojom a cestou, kedy sa výpis ich sady pravidiel načíta knižnicou `yaramod`, premenuje (čo automaticky ovplyvní aj referencie) a novo získané texty pravidiel sa priradia schémam pravidiel.
- `set_privacy` — pravidlám, ktorých ID bolo v inicializačnom zozname, sa odstráni modifikátor `private`, ak je prítomný. Ostatným pravidlám je tento modifikátor pridaný, ak ho už nemali.
- `unset_global` — všetkým pravidlám je odobraný modifikátor `global`, pretože by mal nežiadúci efekt v rámci súvislého textu, kde sú začlenené všetky pravidlá.

- `get_ruleset` — pomocou internej inštancie `RulesetBuilder` získa kompletný `Ruleset` pre pravidlá, ktoré boli zvolené.

5.2 Backend

Časť backend funguje na princípe serveru, ktorý spracúva prichádzajúce HTTP požiadavky. Spracovanie požiadavkov zahŕňa prácu z databázou a prípadne odoslanie RabbitMQ správ. To znamená, že okrem zaisťovania funkcií HTTP serveru, backend definuje modely databázy a prácu s ňou.

Databáza

Pre prácu z databázou bola zvolená Python knižnica `sqlalchemy`[12]. Táto knižnica umožňuje zadefinovať objekty databázy a následne s nimi jednoducho pracovať. Podporuje ORM — object-relational mapping (objektovo relačné mapovanie) — čo znamená, že riadky relačnej databázy sú mapované na objekty programovacieho jazyka. Takto vytvára vrstvu abstrakcie nad samotnou databázou.

Ako sú zadefinované objekty databázy možno vidieť na obrázku 5.4. Miesto atribútu `lineage_id` používaným schému pravidla má databázový model pravidla atribút `oldest_ancestor_id`. Rozdiel je v tom, že tento databázový atribút v prípade pravidla bez predka nemá nastavenú hodnotu, čo je čisto z implementačných dôvodov. Pre modely boli implementované aj statické metódy, ktoré zabezpečujú vykonávanie všetkých potrebných interakcií s databázou.

Na vytvorenie potrebných databázových schém v konkrétnej databáze nie je použitá priamo knižnica `sqlalchemy`, ale Pythonový nástroj `alembic`[13], pomocou ktorého je možné automaticky generovať databázové migrácie pre `sqlalchemy` modely a spúšťať ich. Migrácia je kód popisujúci aké príkazy je nutné vykonať pri prechode zo staršej verzie schém databázy na novšiu verziu, a tiež príkazy potrebné pri prechode opačným smerom. Takto je zabezpečená korektná práca s databázou aj pri úpravách jej modelov počas vývoja.

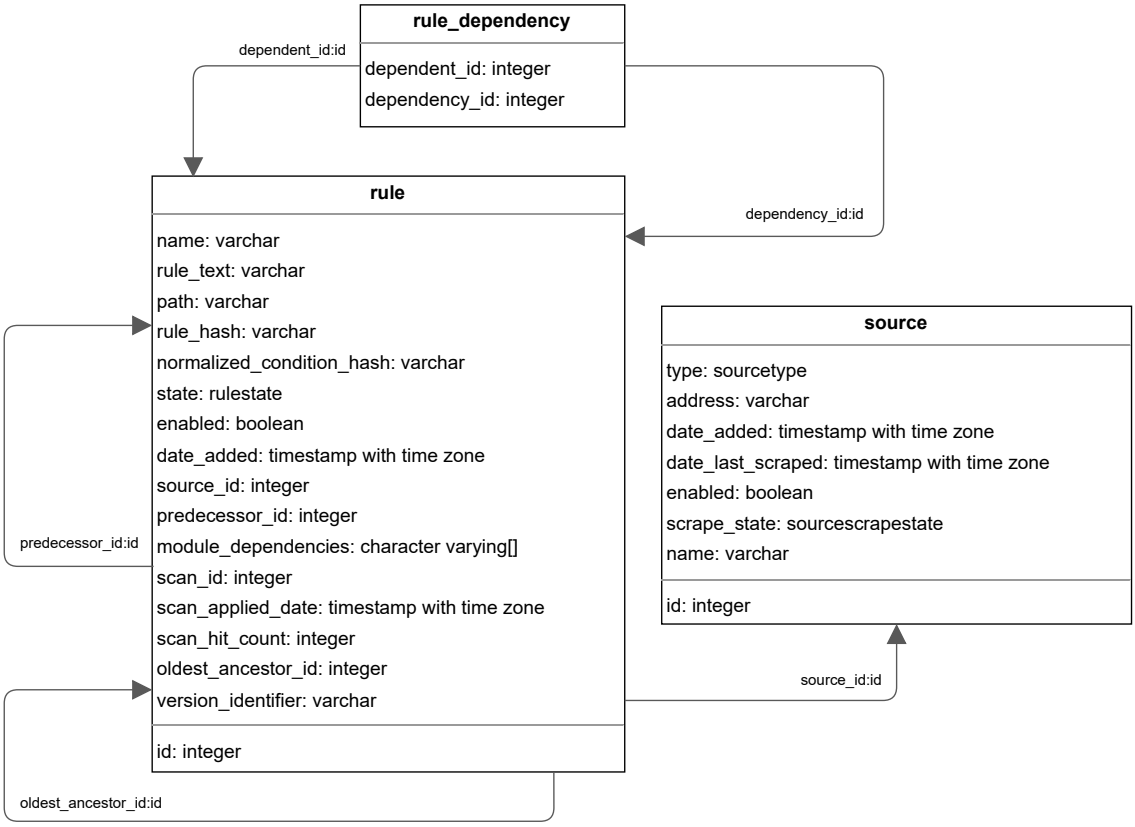
HTTP API

HTTP API bolo implementované za pomoci knižnice `FastAPI`[14]. Podobne ako schémy použité na komunikáciu, aj HTTP API je rozdelená na viacero častí. Časť zameraná na zdroje umožňuje ich čítanie, pridávanie a aktiváciu/deaktiváciu. Pravidlá je možné čítať a aktivovať/deaktivovať. Ďalej je možné spustiť zber, sken a začleňovanie. Pre tento účel má server vytvorené RabbitMQ pripojenie a *exchange*. Pre všetky tri služby je tiež možné na server poslať výsledok danej činnosti. Pri skenoch je navyše možné bezprostredne po započatí skenu poslať notifikáciu priraďujúcu ID daného skenu konkrétnym pravidlám. Taktiež je toto priradenie možné manuálne zrušiť.

Samotné správanie serveru po prijatí požiadavku obnáša iba zavolanie vhodnej statickej metódy (alebo metód) modelu, prípadne funkcie na rozoslanie RabbitMQ správ. Tieto správy sa posielajú, ak sa má začať jedna z troch predom spomínaných činností systému.

Formát zvolený pre serializáciu dát je JSON. V prípade začatia zberu alebo začleňovania sa posielajú všetky relevantné zdroje, pre každý zdroj jedna správa, čo umožňuje distribú-

[12] https://www.sqlalchemy.org/
[13] https://alembic.sqlalchemy.org/en/latest/
[14] https://fastapi.tiangolo.com/



Obr. 5.4: Diagram databázy. Primárne kľúče sú uvedené pod čiarou (`id`). Cudzie kľúče odkazujúce na primárne kľúče sú uvedené pri vzťahoch.

ciu práce pri behu viacerých inštancií danej služby. Službe scanner sa posielajú všetky relevantné pravidlá v jednej správe, aby spustil iba jeden spoločný sken. Ďalej je zabezpečené, že pri prijímaní výsledku nejakej činnosti, ak je táto činnosť kompletne dokončená, sa automaticky spustí nasledujúca nadväzujúca činnosť. Aplikovanie posledného zberu vedie k poslaniu žiadosti o sken a aplikovanie výsledku skenu pošle žiadosti o začleňovanie.

5.3 Scraper

Táto časť funguje na princípe konzumenta RabbitMQ správ. Deklaruje rad (zdieľaný všetkými inštanciami), ktorý príma správy žiadostí na zber z časti backend. Pri prijatí takejto správy sa zo správy získa Python schéma reprezentujúca zdroj, pre ktorý má nastať zber. Pred popisom samotného priebehu zberu táto kapitola ešte obsahuje informácie o knižnici `spire`, svojich triedach a zásuvných moduloch, ktoré sú pri zbere využívané.

Knižnica `spire`

Umožňuje normalizovať a porovnávať pravidlá. Vo veľkej miere využíva knižnicu `yaramod` na spracovanie vstupu a normalizáciu podmienok. Knižnica `spire` je rozdelená na dve časti. Prvou je sada normalizačných tried a tá je využívaná druhou časťou priamo zodpovednou za normalizáciu a porovnávanie.

V rámci svojej implementácie, knižnica `spire` definuje triedu `YaraRule` reprezentujúcu pravidlo a triedu `Match` reprezentujúcu výsledok porovnania. Pri inicializácií objektu `YaraRule` sa vstupný reťazec spracuje knižnicou `yaramod` a na základe výsledku sa nastavia atribúty `YaraRule`. Tieto atribúty majú formu základných Pythonových typov a štruktúr (reťazec, `dict`, atd.), s výnimkou podmienky, kde je využitý priamo yaramod AST. Taktiež obsahuje normalizačnú funkciu `normalize`, ktorá púšťa zoradené čiastkové normalizácie spomínané v ďalšom odseku. Objekty triedy `Match` sa inicializujú pomocou dvoch objektov `YaraRule`, ktoré sa majú porovnať. Pri inicializácií sa jednoducho porovnávajú jednotlivé časti pravidla operátorom `==` (pri podmienke sa porovnáva textová reprezentácia), na základe čoho sa nastavia jednotlivé atribúty objektu. Každý z týchto atribút reprezentuje danú čiastkovú zhodu.

Čiastkové normalizácie sú vykonávané pomocou normalizačných tried, pre ktoré je vytvorená rodičovská trieda `Normalizer` definujúca metódu `normalize`. Táto trieda sama o sebe dedí z `ModifyingVisitor` knižnice `yaramod` a jej metóda `normalize` iba zavolá metódu `modify` (avšak niektoré normalizátory ju predefinujú). Konkrétne normalizačné triedy sú nasledujúce (a v takom poradí sa aj spúšťajú):

1. `ThemNormalizer` — konvertuje výskyty `them` na výraz `($*)`.

2. `WildcardNormalizer` — konvertuje identifikátory reťazca obsahujúce zástupný znak zoznamom konkrétnych zodpovedajúcich identifikátorov. Pri inicializácií vyžaduje `dict` reťazcov pravidla.

3. `StringIdNormalizer` — premenuje identifikátory reťazca v podmienke na názov, ktorý je získaný hexadecimálnym zakódovaním hodnotovej časti (vrátane modifikátorov) daného reťazca. Pri inicializácií vyžaduje `dict` reťazcov pravidla, ktorý je počas normalizácie taktiež modifikovaný.

4. `RuleIdNormalizer` — nahradí identifikátory pravidiel v podmienke podmienkami daných pravidiel. Pri inicializácií vyžaduje `dict` reťazcov pravidla, ktorý je počas normalizácie taktiež modifikovaný, aj yaramod objekt, ktorý obsahuje všetky potenciálne závislosti. Pred vložením získanej podmienky normalizuje túto podmienku tromi predošlými normalizátormi, aby nedošlo ku kolíziám identifikátorov reťazcov s rôznou hodnotou. Okrem týchto troch normalizácií je nad vkladanou podmienkou ešte vykonaná táto normalizácia, čo rieši situácie kedy má závislosť vlastné závislosti. Objekt obsahujúci reťazce pravidla je upravený tak, aby obsahoval novo pridané reťazce.

5. `ForStringNormalizer` — nahradí reťazcové cykly zjednodušeným výrazom `<amount> of <iterable>`, ak je to možné bez zmeny sémantiky.

6. `OfNormalizer` — nahradí výrazy `all of <iterable>` alebo `any of <iterable>` ekvivalentnou logickou konjunkciou alebo disjunkciou.

7. `StringIndexNormalizer` — pridá explicitné indexy pri reťazcových identifikátorov, ak boli použité implicitné.

8. `BoolNormalizer` — normalizuje boolove výrazy na najjednoduchšiu konjunktívnu normálnu formu. Využíva knižnicu `sympy`[15]. Funguje tak, že začne prechádzať boolový výraz od koreňa, konvertujúc operátory `and`, `or`, `not`, literály `true`, `false` a zátvorky

[15] https://www.sympy.org/en/index.html

na ekvivalentné objekty knižnice `sympy`. Ak narazí na niečo iné, celý podstrom reprezentuje logickou premennou (pričom k tejto premennej sa priradí daný podstrom interným asociatívnym poľom). Následne zavolá funkciu spomínanej knižnice `to_cnf`, ktorá zabezpečí zjednodušenie a prevod do konjunktívnej normálnej formy. Rovnocenné výrazy sú touto funkciou zoradené deterministicky. Potom od koreňa prechádza strom výrazov `sympy` a vykonáva spätnú konverziu na `yaramod` výrazy, pričom premenné nahradzuje asociovaným podstromom. Toto sa individuálne vykoná pre každý spracovaný boolovský výraz. Výrazy, ktoré sú priamou súčasťou väčšieho výrazu, nie sú spracované osobitne. Podľa informácií knižnice `sympy`, funkcia `to_cnf` pri zjednodušovaní využíva Quine-McCluskey algoritmus, ktorý ma exponenciálnu zložitosť vzhľadom na počet rôznych premenných. Normalizácie výrazov s počtom rôznych premenných viac ako 8 je nutné dodatočne explicitne povoliť kvôli tomu, že môžu trvať príliš dlho. Pre príklad, pri testovaní času potrebného na spracovanie výrazu $((\ldots \wedge D \wedge C \wedge B \wedge A) \vee (A \wedge \neg B \wedge C \wedge \neg D \wedge \ldots))$ trvalo volanie `to_cnf` pri 8 premenných $\approx$ 1s, pri 9 premenných $\approx$ 8s, pri 10 premenných $\approx$ 70s. `BoolNormalizer` je explicitne povoľuje výrazy s viac ako 8 premennými, ale interne sa riadi vlastným konfigurovateľným limitom, ktorý je atribútom triedy a má predvolenú hodnotu nastavenú rovnako ako knižnica `sympy` na 8. V prípade kedy výraz obsahuje viac ako povolený počet rôznych premenných sa táto normalizácia pre daný výraz preskočí a vypíše sa varovanie.

Triedy

Pre prácu s pravidlami scraper definuje vlastné triedy. Sú to `CurrentRule` a `NewRule`, ktoré zdieľajú rodičovskú triedu `Rule` (odlišná od schémy `Rule`). Trieda `CurrentRule` je určená pre pravidlá, ktoré systém v danom momente už zozbieral v minulosti. Trieda `NewRule` reprezentuje novo spracovávané pravidlá získané priamo zo zdroja. Tiež je zadefinovaná trieda `Match` vyjadrujúca výsledok porovnania dvoch pravidiel.

Spoločný predok `Rule` okrem základných zdielaných atribútov definuje nepovinný atribút reprezentácie daného pravidla objektom `spire YaraRule`. Závislosti odkazujú na iné pravidlá priamo. Samotná trieda má atribút objektu `RulesetBuilder`, ktorý sa využíva pri získavaní textu z kontextom (závislosťami a modulmi) pre dané pravidlo. Definuje aj metódu `is_identical`, ktorá vracia, či je pravidlo identické z iným pravidlom. Pod identitou sa rozumie rovnaký text, cesta a zároveň rekurzívne identita medzi závislosťami.

Pre triedu `CurrentRule` je definovaná metóda zabezpečujúca konverziu zo zoznamu pravidiel reprezentovaných objektami schémy. Trieda `NewRule` zase poskytuje prevod na schému na vytvorenie pravidla, ďalej tiež metódu, ktorá získa objekty `NewRule` zo zvoleného súboru, s odfiltrovaním pravidiel vyvolávajúcich varovania (v Avaste sú nežiadúce). Na získavanie zoznamu závislostí a modulov pravidla bola v rámci triedy `NewRule` definovaná trieda `NewRuleDependencyGetter` (dediaca z `yaramod.ObservingVisitor`), ktorá prejde podmienku pravidla a na základe toho pravidlu k závislostiam priradí asociované pravidlo alebo modul. Keďže sa objekty pravidiel odkazujú na iné pravidlá, získavanie závislostí musí nastávať po skupinách, kde skupinu tvoria pravidlá získané z jedného súboru, aby každé z týchto pravidiel už malo existujúcu objektovú reprezentáciu, ktorú možno priradiť.

Trieda `Match` obsahuje objekt rovnako pomenovanej triedy `Match` z knižnice `spire`, plus výsledok porovnania ciest pravidiel. Za validnú zhodu sa považuje zhoda, kedy aspoň podmienka je zhodná. Sprístupňuje tiež vlastnosť `value`, čiže hodnota, ktorá je reprezentovaná usporiadanou n-ticou prvkov, od najpodstatnejších zhôd po tie najmenej — zhoda

podmienky, cesty, mena, meta informácií, značiek, prítomnosti modifikátora `global` a modifkátora `privát e`. Pri normalizácií sa reťazce upravia tak, že ak je podmienka zhodná, sú zhodné aj tieto reťazce, a keďže zhoda je validná iba ak je podmienka validná, uvažovať zhodu reťazcov je pre tento účel redundantné.

Zásuvné moduly

Zásuvné moduly sú prístupné v zložke `plugins`. Ako súčasť tejto práce bol vytvorený jeden zásuvný modul, pre zdroje typu `GIT`. Využíva knižnicu `gitPython`[16]. Repozitár stiahne do dočasného priečinku a nad súbormi s príponov `.yar` a `.yara` spúšťa metódu triedy `NewRule` pokúšajúcu sa z nich získať pravidlá.

Spracovanie zdroja pri žiadosti na zber

Prvým krokom je vytvorenie objektu reprezentujúcim výsledok zberu. To vyzerá nasledovne:

1. Inicializuje sa objekt výsledku zberu s nastaveným ID zdroja a časom predošlého zberu, čo je získané zo objektu zdroja.
2. Získajú sa objekty `CurrentRule` zo zoznamu `current_rules` daného zdroja.
3. Podľa typu zdroja sa relevantným zásuvným modulom z lokácie identifikovanej adresou zdroja získajú aktuálne pravidlá reprezentované objektami `NewRule`.
4. Pomocou metódy `is_identical` sa porovnávajú dvojice `CurrentRule` a `NewRule`. Týmto sa získa zoznam ID nezmenených pravidiel použitý v objekte výsledku zberu. Zvyšné pravidlá pokračujú v spracovávaní.
5. Normalizujú sa obe sady objektov pravidiel pomocou knižnice `spire`.
6. Získajú sa objekty `Match` pre každú kombináciu objektu `CurrentRule` a `NewRule`, zoskupené podľa hodnoty zhody. Nevalidné zhody sú odfiltrované. Následne sa prechádzajú skupiny zhôd, v poradí od najkvalitnejších zhôd, po tie najslabšie. Zhody sa pravidlám priraďujú ako definitívne, ak ani jednému z oboch porovnaných pravidiel ešte nebola priradená iná definitívna zhoda, inak sa táto zhoda ignoruje.
7. Podľa nájdených zhôd sa objektom `NewRule` nastaví `predecessor_id`, prípadne táto hodnota ostane nenastavená.
8. Pre výsledok zberu sa nastaví zoznam nových pravidiel, ktorý vznikne konverziou objektov `NewRule` na schémy `RuleCreate`.

Následne sa výsledok odošle na backend. Ak pri získavaní výsledku zdroja nastala chyba, zaznamená sa to do výstupu. V takom prípade sa odošle výsledok indikujúci, že sa žiadne pravidlo nezmenilo, aby ostatné časti mohli nadviazať vykonávaním svojej činnosti.

[16] https://gitpython.readthedocs.io/en/stable/index.html

5.4 Scanner

Tok práce tejto časti funguje na takom istom princípe ako scraper. Rozdielom je to, že okrem žiadostí na sken musí scanner taktiež reagovať na správy o výsledku skenu zo systému YARKA. Z toho dôvodu sa vytvárajú dve RabbitMQ pripojenia, ktoré bežia súčasne. Preto je aj miesto knižnice `requests` použitá knižnica `aio_http`, ktorá je asynchrónna, a teda kým jedna časť čaká na HTTP odpoveď, iná môže pracovať. Pre komunikáciu so systémom YARKA sú zadefinované dodatočné schémy, rovnako za použitia knižnice `pydantic`.

Spracovanie žiadosti na sken

Scanner vyberie aktívne pravidlá čakajúce na sken. V prípade, že takéto pravidlá nie sú, pošle informáciu o neuskutočnenom skene na backend. Inak pravidlá spracuje s použitím všetkých spomínaných metód triedy `SchemaRulesPreprocessor` a získa pre ne sadu pravidiel vo forme objektu `Ruleset`. Následne vytvorí žiadosť na sken, pričom textom, ktorý sa má oskenovať, je daná sada pravidiel konvertovaná na reťazce. Ako sady súborov na oskenovanie sú v žiadosti zvolené sady čistých súborov. Od služby YARKA dostane scanner ID započatého skenu. Toto ID, aj ID pravidiel, pre ktoré bol tento sken spustený, sa odošle na backend.

Spracovanie výsledku skenu

Scanner zo správy o výsledku skenu získa status a ID skenu. Pri úspešnom statuse získa pomocou ID skenu cez HTTP API z YARKY zoznam zhôd medzi pravidlami a skenovanými cieľmi. Vďaka názvu pravidla, ktorý bol pri vytváraní žiadosti na YARKA sken upravený na `_<ID>`, získa počet zásahov pre jednotlivé hodnoty ID pravidiel. Informáciu o týchto zásahoch spolu s ID skenu pošle na backend. V prípade, že by sken z nejakého dôvodu zlyhal, je potrebné problém vyriešiť (napr. deaktiváciou problematických pravidiel) a tento sken cez backend invalidovať.

5.5 Merger

Tok práce tejto časti funguje na takom istom princípe ako scraper. Na komunikáciu s GitHub API je použitá knižnica `PyGithub`. Pre každý zo zdrojov, ktorého pravidlá sú začleňované do Avastu, časť merger udržuje Git vetvu `scraper/<source>/main` (kde `<source>` je meno zdroja). Samotný súbor pravidiel, ktorý bude obsahovať všetky začlenené pravidlá zdroja, má cestu `hunting/third_party/scraper/<source>.yar`. Pri aktualizovaní tohto súboru sa nepoužíva princíp graduálnych zmien, ale celý súbor sa prepíše novo vygenerovaným obsahom, obsahujúcim nielen pravidlá čakajúce na začlenenie, ale aj už začlenené pravidlá. Avšak nový obsah bude generovaný tak, aby sa minimalizovali zbytočné zmeny. Na riešenie konfliktov pri zmenách sa zvolil prístup, ktorý je opísaný v podkapitole 4.4.

Prvým krokom mergeru je nadviazanie spojenia so službou GitHub pre repozitár YARA pravidiel Avastu. Následne sa merger pokúsi získať aktuálny obsah súboru obsahujúci pravidlá pre tento zdroj, z vetvy patriacej tomuto zdroju. Ďalej sa zvolené pravidlá zdroja spracujú objektom `SchemaRulesPreprocessor`. Na rozdiel od časti scanner sa v tomto prípade nezíska priamo sada pravidiel. Miesto toho sa pomocou predom získaného obsahu súboru začlenených pravidiel zoradia pravidlá aktuálne spracovávané tak, aby tie, ktoré boli už začlenené, alebo nahrádzajú svojho predchodcu, boli na tej istej pozícií. Párovanie

pravidiel prebieha pomocou atribúty/meta informácie `lineage_id`. Z takto zoradeného zoznamu sa už môže získať sada pravidiel vo forme objektu `Ruleset` a ten sa konvertuje na text.

Vygenerovaný text sa načíta knižnicou `yaramod` a jednotlivé pravidlá, konkrétne ich meta informácie a meno, sa modifikujú. Modifikácie meta informácií v prvom rade obnášajú pridanie prefixu `original_` identifikátorom pôvodných meta informácií pravidla. Potom tiež pridanie meta informácií, ktoré Avast vyžaduje u `hunting` pravidiel. Nakoniec aj pridanie ďalších meta informácií získaných alebo vytvorených týmto systémom, ako napr. `lineage_id` alebo `path`, s prefixom `scraped_`. Pri premenovávaní pravidiel sa používa formát `<source>_scraped_<ID>_<original_name>`, kde `<source>` je meno zdroja, `<ID>` je ID pravidla a `<original_name>` je pôvodný názov pravidla.

Takto už spracovaný text možno začleniť do úložiska Avastu. Prepísanie pôvodného súboru zdroja sa vykoná commitom v hlavnej vetve pre daný zdroj. Navyše sa na novo vzniknutom commite vytvorí nová vetva `scraper/<zdroj>/<datetime>`, kde `<datetime>` je reťazec obsahujúci aktuálny dátum a čas. Pull request sa vytvorí pre túto druhú vetvu, s cieľovou vetvou `master`.

Na backend je odoslaný výsledok začlenenia obsahujúci ID začleňovaných a vymazaných pravidiel. Zoznam ID vymazaných pravidiel je vygenerovaný priamo zo samotného prijatého objektu schémy zdroja, ktorý poslal backend. Týmto zoznamom merger backendu oznamuje, že už prebehlo začleňovanie s informáciou o ich vymazaní.

Kapitola 6

Testovanie systému pre zber, vyhodnotenie, a aktualizáciu YARA pravidiel

Táto kapitola sa zaoberá testovaním systému a vyhodnotením jeho výsledkov. V prvej podkapitole sú uvedené informácie týkajúce sa testovania častí systému počas jeho vývoja. Druhá kapitola sa zameriava na testovanie dokončeného systému ako celku.

6.1 Testovanie počas vývoja

Pri práci na implementácií systému boli využívané aj princípy TDD — test-driven development (vývoj riadený testami). Vývoj pri využívaní týchto princípov vyzerá nasledovne — napíše sa automatický test, overujúci nejakú jednoduchú čiastkovú funkcionalitu, ktorú by vyvíjaný softvér mal mať, ale ešte nemá. Následne sa napíše časť implementácie zodpovedná za túto funkcionalitu, bez brania ohľadu na kvalitu daného kódu. Cieľom je aby novo napísaný test, ako aj všetky predošlé takto vytvorené testy, prešli úspešne. Posledným krokom je refaktorizácia, ktorá napraví kvalitu kódu, pričom testy musia byť stále úspešné. Tieto kroky sa cyklicky opakujú kým sa nedokončí implementácia [1].

Takéto testy boli vytvárané aj pre knižnicu `spire` aj pre systém ako celok. V koreňových adresároch pre ne existuje adresár `tests` s podadresárom `unit`. Hlbšia adresárová štruktúra odráža adresárovú štruktúru samotnej implementáciea pre každý testovaný zdrojový súbor je vytvorený súbor zodpovedný za testovanie jeho funkcií a tried. Dokopy bolo vytvorených 34 takýchto testov pre knižnicu `spire` a 27 pre samotný systém.

6.2 Testovanie finálneho systému

Testovanie systému ako celku prebiehalo dvomi spôsobmi. Systém bol testovaný nad reálnymi, aktívnymi zdrojmi tretích strán. Systém sa taktiež testoval s využitím „umelého“ zdroja, ktorý bol účelovo vytvorený a upravovaný tak, aby sa otestovali rôzne situácie, ktoré by mohli teoreticky nastať a ktoré musí systém vedieť zvládnuť.

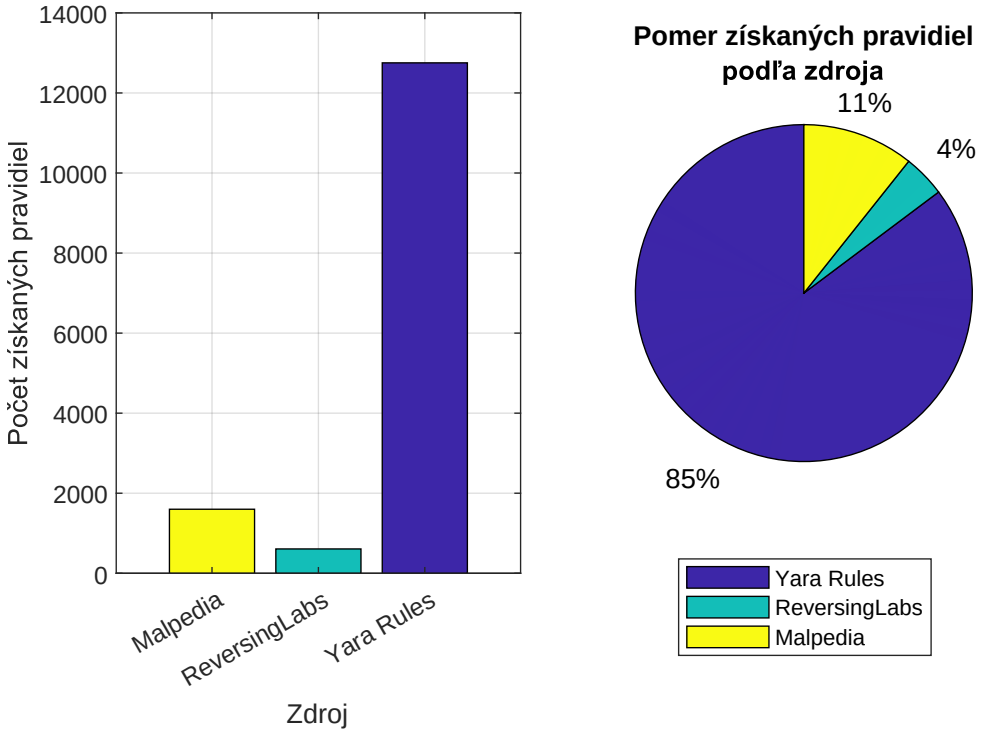
Testovanie nad reálnymi zdrojmi

Ako reálne zdroje boli použité zdroje spomínané v úvode kapitoly 4. V prípade mapedie bol vytvorený Git repozitár, na ktorý bol pravidelne pridávaný súbor obsahujúci všetky verejné pravidlá, získaný pomocou API Malpedie. Pre Malpediu síce existuje aj oficiálny Git repozitár, ale ten je určený iba pre registrovaných užívateľov, a teda obsahuje aj pravidlá, ktoré sa nesmú zverejňovať. Navyše obsahuje aj niekoľko gigabajtov malvérových vzoriek a iných súborov, čo zbytočne zaťažuje sieť a trvá dlhší čas.

Pri prvom zbere sa zozbierali počty pravidiel zobrazené v tabuľke 6.1. Výsledok zberu je tiež vizualizovaný v grafoch na obrázku 6.1.

Zdroj	**Počet získaných pravidiel**	**Podiel získaných pravidiel**
Yara Rules	12752	85,24 %
ReversingLabs	608	4,06 %
Malpedia	1600	10,70 %
Spolu	14960	100 %

Tabuľka 6.1: Výsledky prvého zberu pravidiel zdrojov.



Obr. 6.1: Výsledky prvého zberu pravidiel zdrojov zobrazené formou grafov.

Po zbere nasledovalo skenovanie. Skenuje sa nad sadami čistých súborov, čím sa overuje, či pravidlá vyvolávajú falošné zhody. Pravidlá, ktoré takéto zhody spôsobujú, sú vyhodnotené ako nevyhovujúce pre začlenenie.

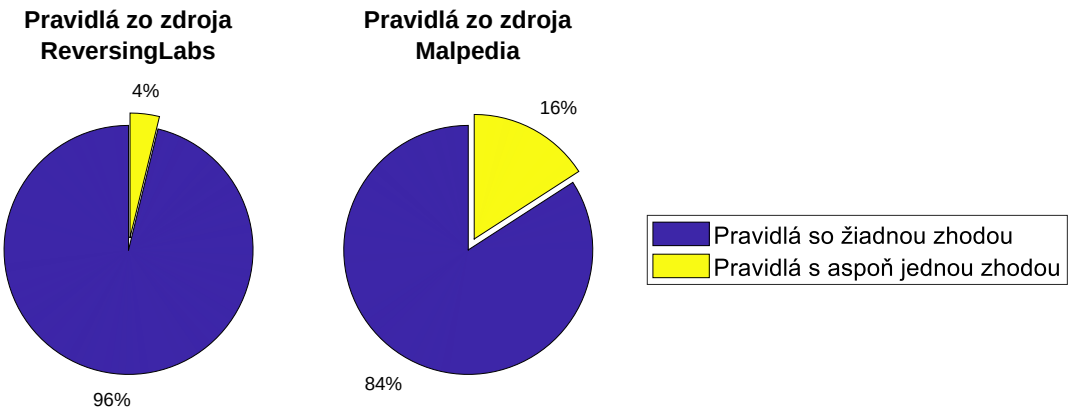
Pri prvom skenovaní týchto pravidiel však nastalo preťaženie systému YARKA. V prvom rade viaceré pravidlá spôsobovali chybové hlásenie „Too many matches“ (príliš veľa zhôd). Toto hlásenie znamená, že konkrétny reťazec pravidla mal v skenovanom súbore priveľa zhôd, čo nadmerne zaťažuje systém. Avšak problémom boli aj nadmerné počty zhôd samotných pravidiel, ktorých bolo niekoľko stoviek miliónov, než sa sken prerušil. Prvé pokusy vyriešiť túto situáciu prebiehali deaktiváciou príliš všeobecných pravidiel. Prvé boli deaktivované pravidlá, ktoré spôsobovali spomínané chybové hlásenie. Potom aj pravidlá, ktorých podmienka pozostávala z jedného reťazca, a pravidlá so značkou `PEiD`, ktorá sa vyskytovala pri obzvlášť nekvalitných pravidlách zo zdroja Yara Rules. Avšak systém bol aj naďalej preťažovaný. Nakoniec sa pristúpilo k úplnej deaktivácií zdroja Yara Rules, ktorý bol identifikovaný ako zdroj väčšiny problematických pravidiel. Týmto sa problém vyriešil.

Z 12752 pravidiel získaných zo zdroja Yara Rules, až 8974 z nich bolo založených na detekcií pomocou jediného reťazca. Zároveň zo 75 pravidiel, ktoré obsahovali reťazec spôsobujúci chybu „Too many matches“ pri prvom skene, 70 patrilo tomuto zdroju. Z týchto údajov aj výsledkov skenovania je vidieť, že tento zdroj ma príliš nízku kvalitu na to, aby bol využitý pre produkčné účely firmy Avast.

Po spomínaných deaktiváciach pravidiel zostalo pre zdroj ReversingLabs aktívnych 605 pravidiel a pre Malpediu 1575 pravidiel. Nad týmito pravidlami bol už vykonaný úspešný sken, na základe ktorého bol vytvorený pull request s pravidlami nespôsobujúcimi zhody. Výsledky tohto skenu sú vidieť v tabuľke 6.2. Výsledok skenu je tiež vizualizovaný v grafoch na obrázku 6.2.

Zdroj	**Počet pravidiel**	**Počet pravidiel so zhodou**	**Počet zhôd**
ReversingLabs	605	23 (3,80 %)	1631
Malpedia	1575	250 (15,87 %)	192 198
Spolu	2180	273 (12,52 %)	193 829

Tabuľka 6.2: Výsledky prvého úspešného skenu pravidiel zdrojov.



Obr. 6.2: Výsledky prvého úspešného skenu pravidiel zdrojov zobrazené formou grafov.

Počas niekoľkotýždňového testovacieho obdobia boli oba zdroje aktualizované práve jeden raz. Do databázy zdroja Malpedia bolo pridané 1 pravidlo a do repozitára ReversingLabs boli pridané 2 pravidlá. Všetky tieto pravidlá prešli úspešne všetkými časťami systému a pre každú z týchto zmien bol vytvorený pull request s očakávanými zmenami.

Testovanie nad umelým zdrojom

Ako možno vidieť v predošlom oddieli, reálne zdroje sú aktualizované pomerne málo často a väčšinou sa jedná o pridávanie, prípadne minimalistické úpravy pravidiel. Preto bol na testovanie využitý aj umelý repozitár, kde boli manuálne zavedené zmeny, ktoré v reálnych repozitároch nastávajú príliš zriedkavo.

Testované boli nasledujúce situácie, v danom poradí:

1. Prvé pridanie umelého zdroja, ktorý obsahuje štyri pravidlá. Jedno z týchto pravidiel využíva modul `PE` a medzi niektorými pravidlami existujú aj závislosti. Pravidlá sú v osobitných súboroch, kvôli závislostiam sa využíva príkaz `include`. Žiadne z týchto pravidiel nevyvoláva falošné pozitíva. Pri začleňovaní by mal byť vytvorený nový súbor obsahujúci všetky pravidlá.

2. Pridanie jedného pravidla (bez falošných pozitív) a odstránenie jedného pravidla zároveň. Novo pridané pravidlo by malo úspešne prejsť skenom a pri začleňovaní by sa malo objaviť na konci súboru. Odstránené pravidlo by malo byť zo súboru vymazané. Poradie ostatných pravidiel musí ostať rovnaké.

3. Modifikácia pravidla zmenou názvu. Pre pravidlo by mal vzniknúť nasledovník, ktorý sa začlení na rovnakú pozíciu, akú malo toto pravidlo.

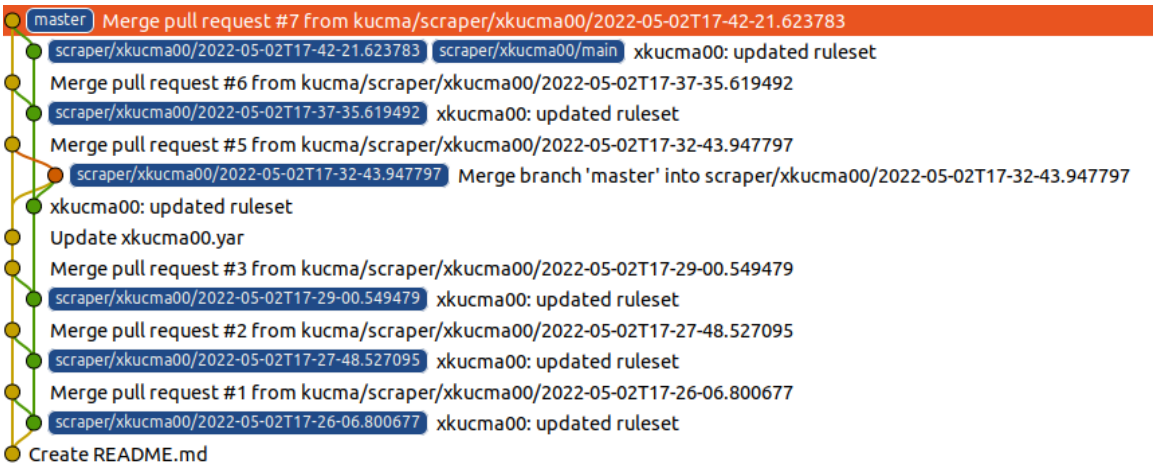
4. Modifikácia pravidla pridaním meta informácie, pričom bola pred tým v cieľovom repozitári vykonaná manuálna modifikácia pridávajúca inú meta informáciu na rovnakú pozíciu. Opäť by mal vzniknúť nasledovník a pri jeho začleňovaní by mal nastať konflikt, ktorý by ale malo byť možné priamočiaro vyriešiť pomocou GitHub grafického rozhrania. Vyriešený bude ponechaním oboch zmien.

5. Modifikácia podmienky pravidla z predošlého bodu so zachovaním sémantického významu. Opäť by mal vzniknúť nasledovník, ktorý sa začlení, pričom by už nemal opätovne nastať konflikt spôsobený modifikáciou z minulého bodu.

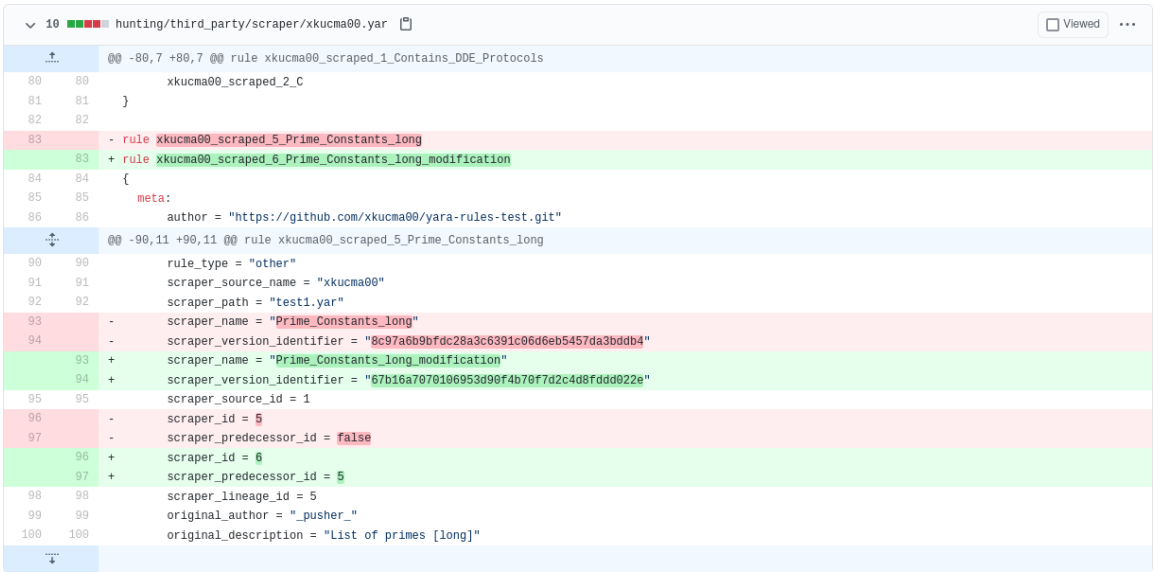
6. Pridanie pravidla, ktoré neprejde testami úspešne. Pre toto pravidlo by nemal vzniknúť pull request.

7. Pridanie `private` pravidla, ktoré prejde testami, a pritom je závislé na pravidle z predchádzajúceho bodu. Toto pravidlo by malo byť začlenené aj spolu s predošlým, pričom predošlé by malo mať pridané modifikátor `private` a nové pravidlo by ho malo mať odstránené.

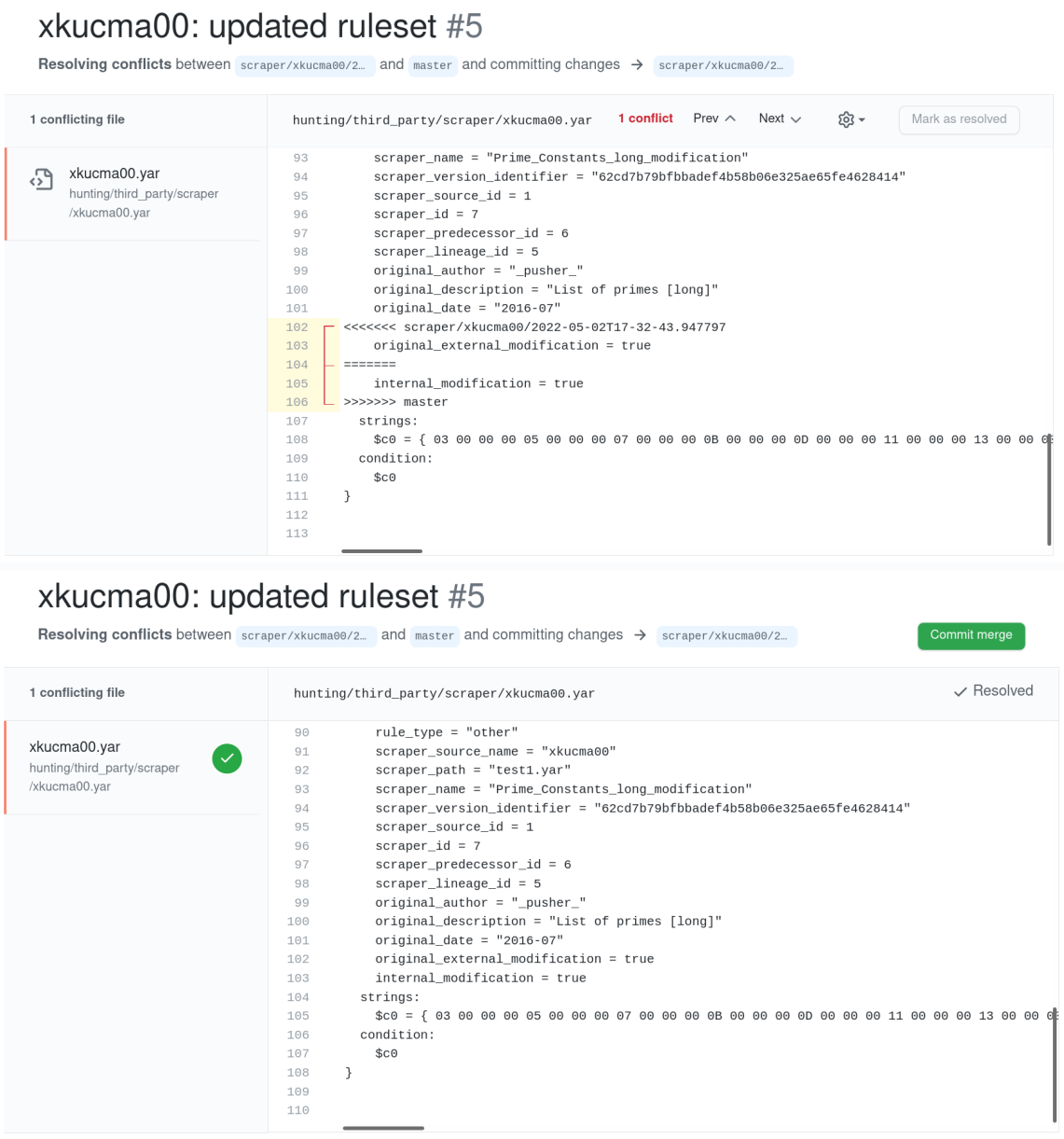
Všetky situácie prebehli podla očakávaní. Vizualizáciu výsledného vetvenia git repozitára po všetkých spomínaných krokoch možno vidieť na obrázku 6.3. Príklad zobrazenia zmien možno vidieť na obrázku 6.4. Riešenie konfliktu z bodu 4 možno vidieť na obrázku 6.5.



Obr. 6.3: Vizualizácia git vetiev vzniknutých počast testovania nad umelým zdrojom. Koreňový commit je najnižšie, najaktuálnejší commit je najvyššie. Vetvy slúžiace na začleňovanie možno po začlenení vymazať, na obrázku sú ponechané kvôli demonštračným dôvodom.



Obr. 6.4: GitHub grafické rozhranie zobrazujúce zmeny pre zdroj Yara Rules



Obr. 6.5: Konflikt pri pull requeste v GitHub grafickom rozhraní pred a po vyriešení

Kapitola 7

Záver

Cieľom tejto práce bolo vytvoriť systém, ktorý by umožňoval analytikom z Avastu jednoduchšie a efektívnejšie využívať voľne dostupné YARA pravidlá tretích strán. Pravidlá by mali byť systémom automaticky zbierané, aktualizované, testované a vo vhodný prípadoch začleňované do úložiska pravidiel Avastu. Tento systém bol úspešne vytvorený.

V prvom rade bolo v rámci tejto práce potrebné oboznámiť sa s fungovaním nástroja YARA, ako aj syntaxou a sémantikov jazyka YARA. Následne boli spracované informácie ohľadom využívania YARY v Avaste. Toto zahŕňalo samotné požiadavky na pravidlá, technológie používanú v súvislosti s nimi, aj ich životný cyklus v rámci Avastu. Potom boli identifikované prvé externé zdroje pravidiel, ktoré by boli potenciálne vhodné pre začlenenie medzi pravidlá Avastu. Z prvých dvoch bodov výskumu vychádzal návrh systému, ktorý by mal byť schopný identifikované zdroje spracovať. Návrh delí daný systém delí na 4 oddelené, ale komunikujúce časti. Na základe tohto návrhu prebehla implementácia samotného systému, pri čom bola vytvorená aj samostatná knižnica na normalizáciu a porovnávanie pravidiel. Vytvorený systém bol otestovaný nad umelo vytvorenými zdrojmi, aj nad tromi reálnymi zdrojmi identifikovanými v rámci jedného z predošlých bodov. Nad reálnymi zdrojmi bol systém testovaný v období niekoľkých týždňov a naplnil očakávania.

Systém bol nasadený v rámci Avastu, kde bude pokračovať v zabezpečovaní spracovaní nových aktualizácií sledovaných zdrojov. Zároveň naďalej poskytuje malvérovým analytikom možnosť začať sledovať a spracovávať nové voľne dostupné zdroje tretích strán, ktoré identifikujú ako vhodné.

Táto práca otvára priestor na nadviazanie a pokračovanie práce na systéme vo viacerých smeroch. Prvým je rozšírenie systému o nové zásuvné moduly, ktoré by umožnili systému pracovať s novými typmi zdrojov. Ďalším možným cieľom vylepšení je práca na knižnici na porovnávanie a normalizáciu pravidiel, ktorá nachádza využitie aj mimo tohto samotného systému. Túto knižnicu by sa dalo skvalitniť pridaním nových čiastkových normalizácií v rámci jej normalizačnej funkcionality, čo by zabezpečilo ešte lepšiu identifikáciu rozdielnych ale sémanticky ekvivalentných pravidiel.

Literatúra

[1] Beck, K. *Test-driven Development: By Example*. 1. vyd. Addison-Wesley, 2003. Addison-Wesley signature series. ISBN 9780321146533.

[2] Beer, B. *Introducing GitHub: A Non-Technical Guide*. 2. vyd. O'Reilly Media, 2018. ISBN 9781491981764.

[3] Guthals, S. a Haack, P. *GitHub For Dummies*. 1. vyd. Wiley, 2019. ISBN 9781119572671.

[4] Klishin, M. et al. *AMQP 0-9-1 Model Explained* [online]. 1 Jan 2022 [cit. 25.4.2022]. Dostupné z: https://www.rabbitmq.com/tutorials/amqp-concepts.html.

[5] Křoustek, J. a Schrötter, M. *YARA@Avast* [online]. 07/10/2021 [cit. 7.1.2022]. Interná dokumentácia Avastu.

[6] Loeliger, J. a McCullough, M. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. 2. vyd. O'Reilly Media, Incorporated, 2012. Oreilly and Associate Series. ISBN 9781449316389.

[7] Milkovič, M. *YARA Infrastructure v2* [online]. September 20, 2019 [cit. 21.1.2022]. Interná dokumentácia Avastu.

[8] Milkovič, M. et al. *Yaramod v3.12.8 documentation* [online]. 7 Apr 2022 [cit. 25.4.2022]. Dostupné z: https://yaramod.readthedocs.io/en/latest/.

[9] Plohmann, D., Clauss, M., Enders, S. a Padilla, E. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *The Journal on Cybercrime & Digital Investigations*. 2018, zv. 3, č. 1. Dostupné z: https://journal.cecyf.fr/ojs/index.php/cybin/article/view/17.

[10] Cuckoo Foundation. *Cuckoo Sandbox - Automated Malware Analysis* [online]. 2019 [cit. 29.12.2021]. Dostupné z: https://cuckoosandbox.org/.

[11] Industrial Control Systems Cybersecurity Emergency Response Team. *Using YARA for Malware Detection*. 2015 [cit. 2.1.2021]. Dostupné z: https://www.cisa.gov/uscert/sites/default/files/Monitors/ICS-CERT_Monitor_May-Jun2015.pdf.

[12] Threat Intelligence Systems. *YARKA* [online]. 13 Apr 2022 [cit. 22.4.2022]. Interný repozitár Avastu.

[13] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. May 1995. Dostupné z: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[14] YaraRules team. *Project* [online]. Wed, Feb 25, 2015 [cit. 22.4.2022]. Dostupné z: https://yara-rules.github.io/blog/project/.

[15] Sikorski, M. a Honig, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1. vyd. No Starch Press, 2012. ISBN 9781593274306.

[16] Szor, P. *The Art of Computer Virus Research and Defense*. 1. vyd. Addison-Wesley, 2005. ISBN 9780321304544.

[17] VirusTotal. *Yara 4.1.3 documentation* [online]. 11 Mar 2021 [cit. 27.12.2021]. Dostupné z: https://yara.readthedocs.io/en/v4.1.3/.

Príloha A

Obsah priloženého pamäťového média

Priložené pamäťové médium má nasledujúcu štruktúru:

- `REAMDE.md` — súbor obsahujúci základné informácie o obsahu priloženého pamäťového média.
- `thesis.pdf` — text bakalárskej práce vo formáte PDF.
- `thesis_print.pdf` — verzia textu bakalárskej práce vo formáte PDF určená na tlač.
- `doc` — adresár obsahujúci zdrojové súbory potrebné na vytvorenie `thesis_print.pdf` a `thesis.pdf`.
- `spire` — adresár obsahujúci knižnicu `spire` vytvorenú v rámci tejto bakalárskej práce, spomínanú v podkapitole 5.3.
- `scraper` — adresár obsahujúci systém vytvorený v rámci tejto bakalárskej práce.