VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

BUDOVÁNÍ EKOSYSTÉMU NÁSTROJŮ YARA-X

DEVELOPMENT OF YARA-X ECOSYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ ĎURIŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. DOMINIKA REGÉCIOVÁ**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

155210

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Ďuriš Tomáš, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Kybernetická bezpečnost
Název:	**Budování ekosystému nástrojů YARA-X**
Kategorie:	Bezpečnost
Akademický rok:	2023/24

Zadání:

1. Seznamte se s projektem a také nástrojem YARA (https://github.com/VirusTotal/yara) sloužící na hledaní vzorů v souborech. Seznamte se také s jazykem YARA, který tento nástroj používá k zápisu vzorů.
2. Seznamte se s nově vznikajícím projektem YARA-X (https://github.com/VirusTotal/yara-x), který v budoucnu nahradí a rozšíří projekt YARA z nástroje pro hledání vzorů na celkovou sadu nástrojů pro analýzu škodlivého kódu.
3. Prostudujte nástroje ekosystému YARA, které vznikly v společnosti Gen (dříve Avast), pro potřeby statické analýzy a ladění (https://github.com/avast/yaramod, https://github.com/avast/yls, https://github.com/avast/yari)
4. Prostudujte nedostatky projektu YARA, které je žádoucí v projektu YARA-X adresovat. Soustřeďte se primárně na oblasti zabývající se laděním (debugovaním) YARA pravidel, jejich parsováním pro potřeby statické analýzy a moduly zpracovávajícími formáty spustitelných souborů. Využijte znalosti z projektů z bodu 3 pro tvorbu jednotného ekosystému.
5. Navrhněte nové nástroje a komponenty v projektu YARA-X pro adresovaní nedostatků z bodu 4. Navržená řešení by měla obsahovat interaktivní nástroj pro vyhodnocování YARA pravidel a framework pro parsovaní odolný vůči chybám pro potřeby použití v integrovaných vývojových prostředích (IDE). Tato řešení diskutujte s konzultantem společnosti Gen (dříve Avast).
6. Naimplementujte navržená řešení z předchozího bodu jako součást ekosystému projektu YARA-X.
7. Implementované řešení otestujte a svoji práci zhodnoťte pro další možný vývoj.

Literatura:

- https://yara.readthedocs.io/en/stable/
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (second ed.). Addison Wesley, Boston. isbn:978-0-321-48681-3
- Rowan: a library for lossless syntax trees. Online: https://github.com/rust-analyzer/rowan
- Swift Syntax and Structured Editing Library. Online: https://github.com/apple/swift/tree/5e2c815edfd758f9b1309ce07bfc01c4bc20ec23/lib/Syntax#swift-syntax-and-structured-editing-library
- Steve Klabnik and Carol Nichols. 2022. The Rust Programming Language, 2nd Edition. isbn: 9781718503106

Při obhajobě semestrální části projektu je požadováno:
Splnění prvních 4 bodů a rozpracování bodu pátého.

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

Vedoucí práce:	**Regéciová Dominika, Ing.**
Konzultant:	Ing. Marek Milkovič
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024

Datum schválení: 30.10.2023

Abstrakt

Cieľom práce je rozšírenie a vytvorenie jednotného ekosystému nástrojov pre jazyk YARA. Práca sa zameriava na pridanie podpory pre moduly slúžiace k získavaniu informácii o štruktúre spustiteľných súborov. Súčasne je predložené vytvorenie modulu slúžiaceho k výpisu získaných informácii a k ich prezentovaniu užívateľovi v rôznych formách. Vytvorené interaktívne prostredie slúžiace k vyhodnocovaniu YARA pravidiel a doplňajúce výsledný ekosystém je riešené pomocou algoritmu umožňujúceho syntaktickú analýzu odolnú voči chybám. Navrhnuté riešenie umožňuje jednoduché napojenie a využitie už existujúcich nástrojov a adresuje nedostatky pôvodného YARA ekosystému. Výsledkom práce je rozšírenie systému o nástroje umožňujúce jednoduchšie ladenie YARA pravidiel, získavanie informácii zo spustiteľných súborov a ich následnú vizualizáciu. Konečné riešenie je riadne otestované, využívané analytikmi a zapracované do hlavnej vetvy projektu YARA-X.

Abstract

The aim of this work is to extend and create an unified ecosystem of tools for the YARA language. The focus is on incorporating modules that can gather information about the structure of executable files. Additionally, a module that can present obtained information to the user in multiple formats is also being proposed. An interactive environment has been created for evaluating YARA rules and enhancing the overall ecosystem by using an error-tolerant parsing algorithm. The proposed solution enables the seamless integration and utilization of existing tools while addressing the limitations of the original YARA ecosystem. The output of the work is an extended system with tools that facilitate the debugging of YARA rules, obtaining information from executable files, and visualizing them. The final solution has been thoroughly tested, utilized by analysts, and integrated into main YARA-X branch.

Kľúčové slová

YARA-X, Rust, Gen Digital, malware, statická analýza, syntaktická analýza odolná voči chybám, Mach-O, extrakcia dát, testovací framework

Keywords

YARA-X, Rust, Gen Digital, malware, static analysis, error-resilient parsing, Mach-O, data extraction, test framework

Citácia

ĎURIŠ, Tomáš. *Budování ekosystému nástrojů YARA-X*. Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Dominika Regéciová

Budování ekosystému nástrojů YARA-X

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Dominiky Regéciovej. Ďalšie informácie mi poskytli Ing. Marek Milkovič a Victor M. Alvarez. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Tomáš Ďuriš
8. mája 2024

Poďakovanie

Rád by som sa poďakoval mojej vedúcej Ing. Dominike Regéciovej za vedenie, odborné poznatky a množstvo cenných rád počas vypracovávania diplomovej práce. Rovnako by som chcel poďakovať konzultantovi Ing. Marekovi Milkovičovi, kolegom a firme Gen Digital za rady, poskytnuté možnosti a nástroje k osobnej aj profesijnej realizácii. V neposlednom rade by som chcel zo srdca poďakovať rodine, kamarátom a predovšetkým priateľke za nekonečnú podporu.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

V súčasnej dobe spojenej s nárastom používania technológii pri každodenných činnostiach predstavuje kybernetická bezpečnosť základný pilier ochrany súkromia na internete. Jedným z potenciálnych útokov voči užívateľovi je využitie škodlivého softvéru (ďalej malvér). Malvér predstavuje bezpečnostnú hrozbu rovnako pre jednotlivcov, ako aj pre menšie alebo väčšie firmy. Je potrebné stanoviť určité pravidlá, akým spôsobom sa pred ním, alebo inými kybernetickými útokmi chrániť. Pokiaľ už k útoku došlo, počítač alebo iné zariadenie bolo infikované a škodlivý softvér máme k dispozícii, je nutné tieto hrozby skúmať a využiť k následnej prevencii. Táto diplomová práca sa zaoberá získavaním a využitím informácii z potenciálne škodlivého softvéru.

Jedným z populárnych nástrojov v tejto sfére je nástroj YARA. Firma Gen Digital Inc., v spolupráci s ktorou je diplomová práca riešená, ale aj veľké množstvo iných antivírusových spoločností využíva tento nástroj na dennej báze. Nástroj YARA predstavuje jazyk, ktorý analytici využívajú k definovaniu pravidiel slúžiacich k detekcii a klasifikácii potenciálne škodlivého softvéru. Tento nástroj bol vytvorený firmou VirusTotal a prvýkrát verejnosti sprístupnený v roku 2013. Každým rokom popularita nástroja YARA narastá a množstvo spoločností, ale aj jednotlivcov využívajúcich tento nástroj sa zväčšuje. Narastá takisto aj množstvo externých nástrojov, ktorého ho, či už priamo alebo nepriamo využívajú. Tieto nástroje tvoria celý ekosystém slúžiaci k zlepšeniu stále sa zhoršujúcej situácie v kybernetickom prostredí. Aj napriek pomerne veľkej komunite vývojárov, ktorí pravidelne prispievajú k modernizácii a vylepšeniu nástroja YARA, existuje stále veľké množstvo priestoru na jeho zlepšenie.

Práca je vyvíjaná formou rozširovania voľne dostupných zdrojových kódov nástroja YARA. Rozšírenia sú priebežne nasadzované do hlavnej vetvy projektu čo môže viesť k tomu, že v novovydaných verziách nastali úpravy predstavených zdrojových kódov. Práca jasne vyznačuje, ktoré časti vznikli ako súčasť práce a akým spôsobom boli upravené. Vzniknuté časti sú vždy konzultované so spoločnosťou Gen Digital a aj s autormi nástroja YARA.

Text diplomovej práce je rozdelený do ôsmich kapitol. Kapitola 2 sa zaoberá predstavením malvéru, jeho analýzou a popisom jednotlivých častí. V kapitole 3 sa práca venuje popisu aktuálneho stavu nástroja YARA a jeho nedostatkom. Nástroj YARA využíva aj množstvo iných nástrojov vyvinutých v spoločnosti Gen Digital. Kapitola takisto obsahuje ich popis v kontexte využitia nástroja YARA a sú navrhnuté možné vylepšenia a prepojenia celého ekosystému. Jedným z vylepšení, na ktorom sa pracuje nezávisle od tejto práce je použitie programovacieho jazyka Rust pre nástroj YARA. Kapitola 4 sa venuje popisu jazyka Rust, prečo je jeho použitie vhodnejšie ako jazyk C, v ktorom bol pôvodne nástroj YARA napísaný. Práca stavia na jeho využití a vylepšení súčasného ekosystému nástrojov. Návrh

implementačnej časti je zahrnutý v kapitole 5 a obsahuje popis jednotlivých rozšírení, ich architektúry a funkcionality. Implementácia je obsiahnutá v kapitole 6 a postupne predstavuje implementovaný modul pre dekompiláciu spustiteľných súborov, modul pre vizualizáciu získaných informácii v rôznych formách zo spustiteľných súborov pomocou nástroja YARA a interaktívne prostredie pre vyhodnocovanie YARA pravidiel za využitia parsera odolného voči chybám. Kapitola 7 popisuje overenie správnosti jednotlivých častí práce. Postupne je popísané testovanie častí kódu a modulov, testovanie generovaním náhodných binárnych vstupov a vytvorenie univerzálneho testovacieho modulu slúžiaceho k dôkladnému otestovanie vstavaných YARA modulov. Práca demonštruje funkcionalitu predstaveného parsera a poukazuje na jeho prínosy voči existujúcim alternatívam. Na záver práce sú v kapitole 8 zhrnuté dosiahnuté výsledky a možnosti ďalšieho rozšírenia do budúcnosti.

Kapitola 2

Analýza malvéru

Pojmom malvér (z anglického malware), ktorý vznikol kombináciou dvoch anglických slov 'malicious' a 'software', označujeme škodlivý softvér [36]. Tento softvér sa vyskytuje vo viacerých formách ako napríklad spustiteľný súbor, skript alebo iný škodlivý softvér určený k napĺňaniu cieľov útočníka. Využíva sa k vykonaniu škodlivej činnosti zahŕňajúcej získanie neoprávneného prístupu k zariadeniu, získanie citlivých osobných údajov alebo iné narušenie súkromia užívateľa. Malvér môže ohrozovať integritu hostiteľského zariadenia, spôsobuje únik citlivých informácii a sprístupňuje ich pre nežiaduce osoby [29].

Existuje viacero spôsobov akými sa malvér vie dostať na hostiteľské zariadenie. Bežným spôsobom ako infikovať zariadenie je presun škodlivého softvéru z infikovaného zariadenia. Nezabezpečené zariadenia na sieti internet sú ľahkým cieľom pre infikovanie bez vedomia užívateľa. Manuálna nákaza môže byť vykonaná podstrčením malvéru a presvedčením užívateľa, že sa jedná o bezpečný softvér, ktorý je následne spustený [36]. Malvér môže využiť zraniteľnosť webového prehliadača k stiahnutiu a následnému spusteniu škodlivého kódu na zariadení obete. Niektoré typy malvéru využívajú bezpečnostné diery a nedostatky v aplikáciach k vykonávaniu škodlivej činnosti. K ochrane používateľov pred škodlivým softvérom sa využívajú programy vyvinuté antivírusovými spoločnosťami. Tieto aplikácie slúžia k prevencii, detekcii a reakcii na malvér. Spôsobom možnej detekcie malvéru sa venuje časť 2.2.

2.1 Rozdelenie malvéru

Na základe rôznych účelov a spôsobov šírenia väčšina typov malvéru spadá do jednej z nasledujúcich kategórii. Triedenie malvéru je dôležité pre identifikáciu nových variánt, pochopenia princípu fungovania jednotlivých druhov a miery rizika, ktorú predstavujú [5].

Vírus: Najznámejší typ malvéru. Dokáže sa rozmnožovať a šíriť do ďalších súborov, aplikácii alebo zariadení. Vírus potrebuje mať hostiteľský program a byť spustený používateľom. Je schopný vykonať deštrukčné činnosti na zariadení. Častým spôsobom šírenia vírusu je pripojenie ho k existujúcemu programu. Následné spustenie programu aktivuje vírus a spustí škodlivú činnosť, ktorá môže byť rôznorodá [36].

Červ: Červ principiálne funguje podobne ako vírus. Rozdiel medzi vírusom a červom je ten, že červ nepotrebuje hostiteľský systém. Červ je schopný samostatného behu a replikácii sa po sieti do iných nezabezpečených zariadení. Niektoré červy slúžia aj ako prostredník na inštaláciu iného typu malvéru [36].

Trojský kôň: Trojský kôň je malvér vložený do aplikácie alebo systému, ktorý na prvý pohľad vykonáva prospešnú činnosť. Typicky obsahuje iný typ malvéru starajúci sa o škodlivú činnosť, ako napríklad zbieranie a odosielanie citlivých informácii útočníkovi [13].

Advér: Účelom advéru je získavanie finančného profitu útočníka pomocou nevyžiadaného prehrávania reklám na postihnutom zariadení. Tento typ malvéru nie je vo svojej podstate škodlivý ale mnoho jeho podôb obmedzuje a vyrušuje užívateľa pri bežnej činnosti. Bežnou praxou je získavanie informácii o užívateľovi a následné prehrávanie personalizovaných reklám [29]. Patrí medzi najrozšírenejšie typy malvéru [35].

Ransomvér: Princíp fungovania ransomvéru spočíva v spustení kryptografického útoku na zariadenie obete. Užívateľovi je odopretý prístup k jeho súborom, pokiaľ nie je zaplatená finančná čiastka útočníkovi. Útočník často tvrdí, že len on je schopný dešifrovať dané zariadenie po útoku [36]. Existuje však množstvo voľne dostupných dekryptorov[1] na známe typy ransomvérov. Jedná sa, podobne ako pri Advéri, o jeden z najpoužívanejších typov malvérov [31].

Spyvér: Spyvér je typ malvéru, ktorý bez vedomia užívateľa získava informácie z postihnutého zariadenia. Útočník môže využiť spyvér k monitorovaniu aktivity užívateľa, zaznamenávaniu ním stlačených kláves alebo k získaniu citlivých informácii, akými sú prístupové heslá alebo finančné a osobné údaje. V minulosti došlo k využívaniu spyvéru pre získanie osobných informácii o používateľoch aj pri celosvetovo známych firmách ako Microsoft, alebo Google [37].

Bot/Botnet: Botnet je škodlivý program umožňujúci útočníkovi získať kontrolu nad zariadením obete. Typický spôsob šírenia botov je pomocou zneužívania bezpečnostných dier v systémoch alebo chýb v aplikáciach. Ak je systém infikovaný, útočník môže doinštalovať iné typy malvéru. Botnet je kolekcia takýchto systémov, ktoré sú spoločne využívané útočníkom k získaniu väčšej výpočetnej sily k vykonávaniu škodlivých útokov. Príkladom takéhoto typu je útok DDoS, využívaný k zahlteniu sieti alebo pamäte zariadenia prostredníctvom odosielania veľkého množstva nevyžiadaných správ [36].

2.2 Detekcia a analýza malvéru

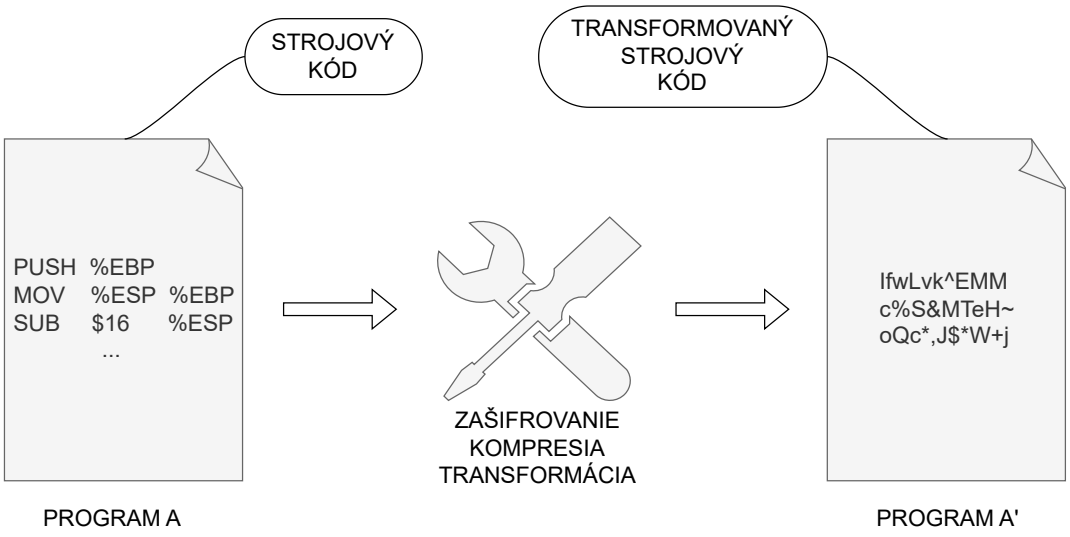
Autori škodlivého kódu používajú viacero techník k sťaženiu detekcie malvéru. K zabráneniu detekcie škodlivého softvéru sú využívané rôzne spôsoby, od jednoduchých a pomerne bežných techník spočívajúcich vo vložení časti kódu do programu, až po zložitejšie techniky využívajúce algoritmus na vytvorenie a zamaskovanie malvéru [29]. Medzi spomínané techniky ochrany malvéru pred detekciou patrí:

- **Šifrovanie -** pozostáva zo šifrovacieho algoritmu, šifrovacieho kľúča, algoritmu na dešifrovanie a zo samotného škodlivého kódu. Šifrovací kľúč spolu so šifrovacím algoritmom sú používané na vytváranie nových verzii malvéru a tým sťaženiu ich detekcie [27].
- **Zabalenie škodlivého kódu -** spočíva v komprimácii veľkosti. Malvéru je zároveň obalený ďalšou vrstvou, ktorá slúži k sťaženiu jeho detekcie. Často používanou technikou je *'rozbalenie za behu'*, kedy v operačnej pamäti dochádza k rozbaleniu zabaleného súboru pri jeho spustení [36].
- **Obfuskácia -** skrýva logiku programu a sťažuje jeho pochopenie pri prvotnej analýze. Táto technika používa metódy, ako pridávanie zbytočného kódu, preusporiadanie re-

[1] https://decoded.avast.io/tag/decryptors/

gistrov, zámenu hodnôt premenných, zakódovanie pomocou base64 a mnoho ďalších [14]. Obfuskácia malvéru je znázorňená na obrázku 2.1.

- **Polymorfný malvér -** je schopný sa prispôsobovať a meniť počas behu. Hlavným účelom je sťažiť jeho identifikáciu a možnú deštrukciu. Polymorfný malvér využíva mutáciu kódu pri každej infikácii. Na identifikáciu polymorfného malvéru sa využíva jeho sémantika, ktorá aj napriek meniacemu sa kódu, ostáva vždy rovnaká [27]. Pomerne nová štúdia ukázala, že viac ako 94% škodlivých spustiteľných súborov je polymorfných [34].
- **Metamorfný malvér -** sa považuje za najkomplexnejší typ škodlivého softvéru. Škodlivý kód, ktorý obsahuje, je schopný sám seba zmodifikovať a nepredstavovať žiadnu podobnosť s pôvodným kódom. Nové verzie kódu sú sofistikovanejšie, ale funkcionalita ostáva rovnaká. Pri polymorfnom a metamorfnom malvéri sú využívane techniky obfuskácie a ich kombinácie k získaniu novej verzie škodlivého kódu obsiahnutého v súbore [27].



Obr. 2.1: Príklad ochrany pred detekciou malvéru.

Kombinácia vlastností popisujúcich malvér sa nazýva podpis. Pred vytvorením podpisu je nutné škodlivý kód analyzovať a pochopiť ako funguje a aké riziku predstavuje. Na analýzu škodlivého programu sú využívané techniky rozdelené do dvoch základných kategórii a to na statickú a dynamickú analýzu [12].

Statická analýza

Statická analýza spočíva v skúmaní programu bez jeho spustenia. Prvky využívané pri statickej analýze sú reťazce, podpisy, vložené symboly, sekvencie bytov, použité operačné kódy a iné [12]. Statická analýza nám umožňuje analyzovať program vo viacerých formách bez ohľadu na architektúru, alebo operačný systém. Ďalšou z výhod použitia statickej analýzy

je jej bezpečnosť, keďže nevystavuje užívateľa rizikám spojeným so spustením programu. Ak je k dispozícii zdrojový kód programu, sme schopní priamo aplikovať nástroje na statickú analýza a nájsť potenciálne chyby a riziká programu. Vo väčšine prípadov je však dostupná len binárna reprezentácia programu. Programy sú v počítači reprezentované v binárnej podobe, tá umožňuje pracovať s nimi rýchlo a efektívne. Táto podoba však len ďalej sťažuje analýzu programu, keďže dochádza k strate typových informácii.

Statická analýza typicky zahŕňa analýzu formátu spustiteľného súboru. Umožňuje analytikovi získať prehľad o jeho štruktúre, rozdelení na dáta a kód, a identifikáciu importovaných či exportovaných symbolov. Druhým spôsobom ako staticky analyzovať binárne súbory môže byť *'dissassembling'*. Tento anglický pojem opisuje činnosť, ktorá umožňuje spätne rekonštruovať kód a transformovať inštrukcie v binárnej podobe do jazyka symbolických inštrukcií. Pomocou nich je možné získať väčší prehľad o tom, ako program funguje a odhaliť poznávacie znaky využívané k odhaleniu útočníka [13]. Často využívaným prvkom statickej analýzy je skúmanie toku funkcií a konštrukcia grafu znázorňujúceho beh programu [36]. Ďalším spôsobom môže byť dekompilácia zdrojového kódu, ktorá slúži na získanie pôvodnej reprezentácie programu. Zdrojový kód môže poskytovať množstvo užitočných informácií obsahujúcich napríklad navštívené webové stránky, alebo volanie potenciálne nebezpečných knižníc.

Voči statickej analýze je pomerne jednoduché sa chrániť. Spôsoby ochrany zahŕňajú napríklad použitie šifrovania, obfuskácie, zabalenie programu alebo použitie polymorfného malvéru. Autori malvéru, rovnako ako aj analytici zaoberajúci sa ním poznajú limity tejto technológie a vedia ich využiť vo svoj prospech. To podmieňujú použitie alternatívnych prístupov, ktoré spadajú pod dynamickú analýzu. Nevýhodou statickej analýzy je aj fakt, že výsledný program a jeho beh môže byť vo veľkej miere ovplyvnený rôznymi vonkajšími faktormi ako vstup, hardvér alebo sieť. Statická analýza síce umožňuje preskúmať všetky možnosti vetvenia a toku kódu, ale ľahko sa môže stať, že výsledný program sa správa inak ako očakávame [11].

Jedným z rozšírených nástrojov pre statickú analýzu je nástroj YARA, ktorý umožňuje vytvárať pravidlá pre popis vlastností malvéru, podpisov, na základe textových reťazcov alebo binárnych vzorov, ktoré program obsahuje [24]. Nástroju YARA, jeho špecifikácii, použitiu, nedostatkom a možným vylepšeniam sa venuje kapitola 3.

Dynamická analýza

Narozdiel od statickej analýzy, dynamická analýza skúma škodlivý softvér počas jeho behu. Program je analyzovaný v plne kontrolovanom prostredí, ako napríklad na virtuálnom stroji, či za použitia emulátorov. Princíp kontrolovaného prostredia spočíva v tom, že program nemá prístup mimo toto prostredie a minimalizuje sa riziko pri spúšťaní škodlivých alebo neznámych programov. Poskytnutie informácii o behu programu je zabezpečené pomocou prítomnosti veľkej škály monitorovacích nástrojov, ako napríklad Wireshark, Process Explorer, Process monitor alebo Regshot [12]. Dynamická analýza je možná aj pomocou využitia nástroja YARA pomocou skenovania dynamických rysov voči získaným záznamom z izolovaného prostredia. Využíva sa Cuckoo modul a Cuckoo sandbox, ktorý umožňuje spustiť softvér v bezpečnom prostredí a získať informácie o jeho správaní.

Dynamická analýza umožňuje obísť problémy vyskytujúce sa pri statickej analýze ako napríklad obfuskácia kódu. Kód je možné vidieť priamo za behu vďaka čomu je možné pochopiť ako presne daný malvér funguje. Techniky využívané pri dynamickej analýze zahŕňajú ladenie, monitorovanie volaní funkcií, analýza behu programu alebo parametrov

funkcií. Ladenie využíva špeciálny program nazývaný *'debugger'*, ktorý umožňuje skúmať práve spustenú časť kódu, alebo pozastaviť beh program a získať obsah registrov a hodnoty premenných. Pokročilé typy malvéru môžu obsahovať nástroje, ktoré umožňujú identifikovať, že je malvér spustený v kontrolovanom prostredí alebo pomocou debuggera. Následne kód je schopný zmeniť svoje správanie alebo úplne zastaviť škodlivú činnosť a tým sa vyhnúť jeho odhaleniu a analýze [4]. Na zaznamenávania a zobrazovania volaných funkcií slúži technika s názvom monitorovanie volaní funkcií. Okrem iného pomáha analytikovi identifikovať kľúčové funkcie využívané malvérom ku škodlivej činnosti a je úzko spojená s analýzou behu programu. Analýza behu programu je schopná odhaliť dynamické vlastnosti skryté pred statickou analýzou. Monitorujú sa registre, súborový systém, procesy, vlákna a iné. Skúmanie parametrov funkcií pri statickej analýze dokáže odvodiť množinu možných hodnôt. Dynamická analýza dokáže túto funkcionalitu povýšiť o získanie skutočných hodnôt parametrov. Umožňuje získať prehľad funkcií, ktoré operujú nad rovnakým objektom, aký vstup očakávajú a čo generujú [11].

Prirodzené správanie malvéru je spravidla odolnejšie voči statickej analýze. Dynamická analýza je efektívnejšia, ale zároveň robustnejšia a náročnejšia na čas a zdroje. Kontrolované prostredie nie je vždy totožné s reálnym prostredím, v ktorom sa malvér šíri čo častokrát vedie k neprirodzenému a zavádzajúcemu sa správaniu. Dynamická analýza je častokrát veľmi účinná pri analýze zabalených súborov. Tento typ malvéru vyžaduje pri svojom behu v určitý moment rozbalenie a následne je samotný kód dostupný len v operačnej pamäti. Dynamickú analýzu je nutné vykonávať v dostatočne izolovanom prostredí a minimalizovať mieru rizika, ktorú predstavuje spúšťanie infikovaných súborov [4].

V praxi sa využíva kombinácia statickej aj dynamickej analýzy. Každá z nich obsahuje svoje silné a slabšie stránky. Statická analýza môže slúžiť na prvotnú a efektívnu identifikáciu a klasifikáciu malvéru pomocou jeho vlastností - podpisov. Dynamická analýza je následne schopná presnejšej analýzy správania sa konkrétneho typu. V prípade komplexnejšieho typu malvéru je možné využiť prístup, kedy je pri zabalenom malvéri využitá dynamická analýza k odhaleniu a extrakcii skrytých častí kódu za behu programu. Statická analýza je následne použitá na podrobnejšiu analýzu odhalených častí [36]. Kombinácia oboch prístupov k presnejšej detekcii a eliminácii malvéru je využívaná v akademickej, ale aj komerčnej sfére v podobe antivírusových programov.

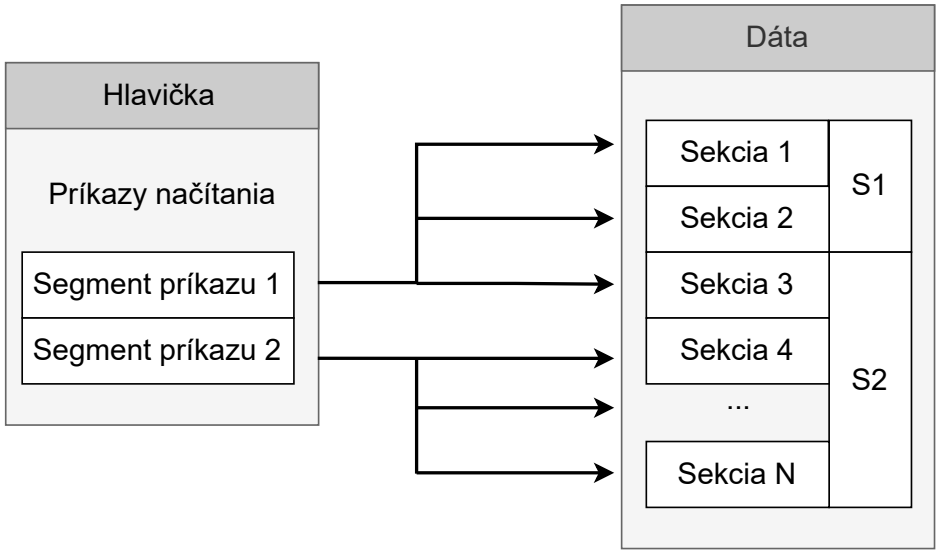
2.3 Formát spustiteľných súborov

Za spustiteľný súbor sa považuje druh počítačového súboru, ktorý obsahuje kód spolu s dátami potrebnými pre jeho spustenie na danom operačnom systéme. Formát uloženia dát sa líši systém od systému. Jedná sa o súbor využívaný k vykonaniu rôznych funkcií alebo operácii nad zariadením. Na rozdiel od bežného súboru, spustiteľný súbor nie je možné jednoducho prečítať, keďže je skompilovaný. Obsahuje v sebe inštrukcie vykonávané operačným systémom. Môže byť špecifický pre jeden konkrétny operačný systém alebo môže byť kompatibilný s viacerými systémami.

Spolu s množstvom výhod spustiteľné súbory predstavujú aj určité nevýhody. Vzhľadom na ich náročnú čitateľnosť a reprezentáciu je pomerne ľahké ukryť v nich nečakané prípadne škodlivé chovanie. Malvér je možné skryť v spustiteľných súboroch využitím už spomínaných techník v kapitole 2.2, ako napríklad zabalením alebo šifrovaním súborov.

K odhaleniu malvéru je možné využiť prvky statickej analýzy. Podpis predstavujúci sekvenciu bytov v súbore, ktorá je špecifická pre konkrétny typ malvéru, je často využívaným prvkom. Pomocou nástroja YARA je možné definovať vzory založené na podpisoch alebo iných jedinečných prvkoch popisujúcich špecifickú časť súboru. Následne prebieha analýza spustiteľného súbora a vyhľadávanie vzorov v jeho častiach. Medzi najpoužívanejšie formáty patria [9]:

- **P**ortable **E**xecutable - Formát spustiteľného súbora špecifický pre operačný systém Windows. Umožňuje spustenie súboru na viacerých typoch procesorov. Hlavička PE súboru obsahuje povinnú DOS hlavičku pozostávajúcu zo zakódovaného reťazca *'MZ'*. Sú to iniciály mena Mark Zbikowski, vývojára spoločnosti Microsoft, ktorý sa pričinil o vznik MS-DOS. Časť predstavujúca DOS hlavičku je spustená pokiaľ bol súbor spustený na DOS systéme a vo väčšine prípadov obsahuje len hlášku, že je potrebné tento súbor spustiť na Win32 systéme. Nasleduje PE hlavička, ktorá jednoznačne identifikuje, že sa jedná o PE súbor [22].
- **E**xecutable and **L**inkable **F**ormat - Využívaný operačným systémom Linux. Používa sa napríklad pre spustiteľné súbory alebo zdieľané knižnice. Rovnako ako PE súbor aj ELF súbor umožňuje spustenie na viacerých typoch procesorov. ELF súbor pozostáva z hlavičky a sekcie dát. Prvé 4 byty obsahujú prefix **0x7F** a následne zakódovanú hodnotu *'ELF'*. Táto časť je povinná a určuje, že sa skutočne jedná o ELF súbor [8].
- **Mach-O** - Skratka pre *'Mach object'*. Mach-O súbory sa používajú na systémoch založených na Mach kernely, ako napríklad macOS a iOS. Tento typ spustiteľného súboru obsahuje hlavičku, zoznam príkazov načítania a príslušné vnorené sekcie. Hlavička Macho súboru obsahuje rovnako ako predošlé dva prípady špeciálnu hodnotu potvrdzujúcu, že sa jedná o Macho súbor. Táto hodnota takisto určuje endianitu a architektúru zariadenia [3]. Schéma 2.2 popisuje základnú štruktúru Mach-O formátu.



Obr. 2.2: Formát spustiteľného súboru typu Mach-O.

Spomenuté formáty predstavujú väčšinu používaných formátov spustiteľných súborov. Každý z nich má svoju špecifickú štruktúru, ktorá sa však dá generalizovať do dvoch hlavných častí, na hlavičku súboru a vnorené sekcie [32]. Hlavička súboru je blok metadát obsahujúci základné informácie o súbore a o tom, o aký konkrétny typ súboru sa jedná. Ďalšími nepovinnými časťami hlavičky môže byť veľkosť súboru, adresa vstupného bodu súboru, verzia súboru, pre aké architektúry je daný súbor určený, použité knižnice, symboly a mnoho ďalších podstatných informácii. Hlavička môže rovnako obsahovať popis jednotlivých sekcií spustiteľného súboru a ich uloženie. Sekcie sú špecifické časti spustiteľného súboru, ktoré obsahujú samotný kód alebo využívané zdroje a dáta programu [2]. Praktická časť práce sa venuje zobrazeniu dát zo spustiteľných súborov pre účely nástroja YARA, respektíve jej novovyvíjanej verzie YARA-X. Okrem toho sa zameria aj na samotné získavanie informácii z macho súborov bližšie popísané v sekcii 5.1.

Kapitola 3

Popis jazyka YARA a jeho využitie

YARA je populárny open-source nástroj, ktorý využíva rysy a charakteristické znaky rodín malvérov k ich detekcii a klasifikácii. Jeho skratka je rekurzívny akronym pre *'Yet Another Ridiculous Acronym'* prípadne *'Yet Another Recursive Acronym'* a bol vytvorený spoločnosťou VirusTotal, teraz už spadajúcou pod Google a autorom je Victor M. Alvarez. Dokumentácia nástroja YARA ho popisuje ako nástroj zameriavajúci (ale nie limitujúci) sa na pomoc analytikom identifikovať a klasifikovať druhy malvéru [33]. Pomocou neho vieme využiť textové alebo aj binárne reťazce, obsiahnuté v súboroch k vytvoreniu podpisu rodín malvéru. YARA je charakterizovaná ako multiplatformový nástroj a teda sme je možné ho využívať na všetkých najpoužívanejších operačných systémoch (Windows, MacOS a Linux) [30]. Je napísaný v jazyku C a kladie veľký dôraz na rýchlosť a efektivitu jeho využívania. Nástroj YARA je využívaný veľkým množstvom antivírusových spoločností. Medzi najznámejšie firmy využívajúce tento nástroj patrí: Gen Digital, ESET, Kaspersky, McAfee, alebo iné spoločnosti ako napríklad Cisco.

Okrem nástroja určeného na detekciu a klasifikáciu malvéru sa tento akronym používa aj pre jazyk, ktorý k tomu využíva. Tento jazyk pozostáva z pravidiel vytvorených na základe znalostí autora nadobudnutých pri statickej alebo dynamickej analýze. YARA využíva deklaratívny prístup k hľadaniu definovaných vzorov v pravidlách v spracovávaných súboroch. Bližšiemu popisu YARA pravidiel a ich zloženiu je venovaná sekcia 3.1. Okrem klasifikácie už známych rodín malvéru je možné tento nástroj využiť na detekciu predtým neznámeho typu malvéru. Umožňuje hľadať vzory v nespustenom súbore, ale aj využívať chovanie súboru za behu k ich odhaleniu. Jednoduché pravidlá častokrát zahŕňajú hľadanie konkrétneho reťazca v súbore, naopak zložité pravidlá využívajú behaviorálne prvky súboru a môžu slúžiť k hľadaniu špecifických dát vo virtuálnej adrese spusteného programu.

Nástroj YARA sa skladá z dvoch hlavných častí. Počiatočná fáza prekladá definované pravidlo do *bytekódu*. Takto skompilované pravidlo je následne využité k vyhľadávaniu v ňom definovaných vzorov v predloženom súbore. Preklad pravidla zahŕňa aj syntaktickú a sémantickú analýzu. Vyhľadávanie je možné nad súbormi, adresármi, prípadne aj spustenými procesmi. YARA je primárne dostupná ako program určený pre príkazový riadok, ale poskytuje takisto knižnice pre jazyk Python a C. Rozšírením tohto jazyka sú moduly obsahujúce prídavnú funkcionalitu v podobe nástrojov pre prácu s časom, reťazcami, hashmi, alebo spustiteľnými súbormi (Macho, PE, ELF, LNK a iné). Bližší popis implementovaných modulov pre nástroj YARA je v kapitole 3.2.

3.1 Štruktúra YARA pravidiel

Pravidlá napísané v jazyku YARA slúžia na detekciu malvéra pomocou hľadania zhodných reťazcov medzi tými definovanými v pravidle a tými, čo sa nachádzajú v analyzovanom súbore. Pravidlá obsahujú vopred definované reťazce, ktoré charakterizujú a jednoznačne identifikujú konkrétny typ malvéru alebo jeho rodinu [25]. Syntax napísaného pravidla pripomína syntax jazyka C. Súbor s pravidlami môže obsahovať jedno alebo viacero pravidiel. Pravidlá špecifikované v rámci jedného súboru môžu odkazovať na skôr definované pravidlá v rovnakom súbore. Súbor pravidiel môže tvoriť aj skupina pravidiel, ktorá je na sebe nezávislá a nemusí sa viazať ani k jednému typu/rodine malvéru. Okrem pravidiel môže súbor obsahovať vkladanie modulov alebo súborov s inými pravidlami pomocou kľúčových slov `import`, respektíve `include`.

Každé YARA pravidlo začína kľúčovým slovom `rule`, za ktorým nasleduje názov pravidla špecifikované jeho autorom. Názov pravidla jednoznačne identifikuje pravidlo v rámci daného súbora. Môže obsahovať alfanumerické znaky a podčiarkovník, nesmie však začínať číslicou podobne ako identifikátory v jazyku C. Nasleduje telo pravidla uzatvorené v zložených zátvorkách. Telo pravidla sa delí do dvoch hlavných častí - definícia reťazcov a sekcia podmienok.

Definícia reťazcov

Voliteľná časť pravidla, ktorá obsahuje definície reťazcov použitých v podmienkach. Každý definovaný reťazec obsahuje identifikátor pozostávajúci zo znaku $ nasledovaného sekvenciou kombinácie alfanumerických znakov a podčiarkovníku. Reťazce môžu byť definované v textovej podobne, hexadecimálne podobe alebo v podobe regulárneho výrazu, viz obrázok 3.1.

- Textové reťazce - Predstavujú základný stavebný kameň YARA pravidiel. Jedná sa o textový reťazec uzavretý v zátvorkách, ktorý obsahuje zobraziteľné ASCII znaky obohatené o menšiu podmnožinu escape sekvencií. Môžu byť modifikované pomocou použitia vstavaných modifikátorov, ako napríklad `wide`, `base64`, `nocase` a mnoho iných.

- Hexadecimálne reťazce - Je to množina hexadecimálnych čísiel uzavretých v zložených zátvorkách. Obsahujú štyri špeciálne konštrukcie, ktoré umožňujú väčšiu flexibilitu ich využitia. Patria medzi ne zástupné znaky, umožňujúce nahradenie jedného alebo viacerých znakov. Operátor `not` slúži k špecifikovaniu hodnoty, ktorú daný byte nemá nadobúdať. Ďalšie dve konštrukcie sú alternatívy, umožňujúce špecifikovať viacero variánt pre daný byte alebo skupinu bytov a skoky, ktoré umožňujú 'preskočiť', teda ignorovať skupinu bytov reprezentovaných hexadecimálnou podobou.

- Reťazce obsahujúce regulárne výrazy - Regulárne výrazy sú jeden z najsilnejších vyjadrovacích prostriedkov, ktorými jazyk YARA disponuje. Narozdiel od textových reťazcov, regulárne výrazy sú ohraničené lomkou. Aj napriek ich silnej vyjadrovacej schopnosti a poskytovaniu väčšej flexibility pri písaní pravidiel ich používanie výrazným spôsobom spomaľuje vyhodnocovanie pravidiel a mali by byť používané len ak sú ostatné prostriedky nedostačujúce [28]. YARA obsahuje vlastný nástroj na spracovanie regulárnych výrazov. Obsahuje väčšinu funkcionality knižnice PCRE[1] pre jazyk

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

PERL. Výnimkou sú spätné referencie alebo napríklad znakové triedy `isalpha` alebo `isidigit`. Podrobnejší popis podporovaných regulárnych výrazov je možné nájsť v sekcii `strings`[2], ktorá je súčasťou oficiálnej dokumentácii nástroja YARA.

```
rule moje_pravidlo
{
    strings:
        $textovy_retazec = "retazec"
        $hexadecimalny_retazec = { 00 01 02 03 04 05 }
    condition:
        $textovy_retazec or $hexadecimalny_retazec
}
```

Obr. 3.1: Základné YARA pravidlo s textovým a aj hexadecimálnym reťazcom.

Sekcia podmienok

Obsahuje základnú logiku pravidla. Skladá sa z pravdivostných výrazov určujúcich, či dané pravidlo platí, alebo nie. Napríklad v pravidle na obrázku 3.2 dochádza k vyhodnoteniu pravidla ako pravdivého, pokiaľ sa obsah reťazca s názvom `'textovy_retazec'` alebo `'hexadecimalny_retazec'` nachádza v analyzovanom súbore, ktorý je zároveň menší ako 1MB. Súčasťou pravdivostného výrazu môžu byť:

- logické operátory 'or', 'and', alebo 'not'
- relačné operátory '==', '!=', '<', '>', '<=' a '>='
- aritmetické operátory '+', '-', '*', '/' a '%'
- bitové operátory '&', '|', '«', '»', '~' a '^'

Ďalšou potenciálnou súčasťou podmienky môžu byť zložitejšie konštrukcie slúžiace napríklad k získaniu veľkosti súboru, počítaniu počtu reťazcov, prístupu do určitej časti pamäte alebo k iterovaniu v rôznych sekciách súbora.

[2] https://yara.readthedocs.io/en/latest/writingrules.html#strings

```
rule moje_pravidlo
{
    strings:
        $textovy_retazec = "retazec"
        $hexadecimalny_retazec = { 00 01 02 03 04 05 }
    condition:
        (
            $textovy_retazec or
            $hexadecimalny_retazec
        ) and
        filesize < 100MB
}
```

Obr. 3.2: YARA pravidlo obohatené o logické a relačné operátory.

Okrem spomenutých základných častí pravidlo môže obsahovať aj voliteľné rozšírujúce časti, akými sú:

- Sekcia komentárov - Obsahuje jeden alebo viacero komentárov popisujúcich dodatočné informácie. Tieto informácie sa nevyužívajú, ale slúžia k dodatočnému popisu pravidla. Pri kompilácii sú zahodené.
- Tagy - Slúžia k filtrovaniu pravidiel vo výstupe. Umožňujú zobraziť len pravidlá, ktoré užívateľa zaujímajú. Sú špecifikované za názvom pravidla a oddelené od neho pomocou ':'. Je možné špecifikovať viacero tagov a oddeliť ich medzerou.
- Metadata - Sú určené, podobne ako komentáre, k dodatočnému popisu pravidla s tým rozdielom, že pri kompilácii nie sú zahodené. Sekcia metadát sa v pravidle značí pomocou kľúčového slova `meta` a nasleduje dvojica `<kľúč> = <hodnota>`. Môžu slúžiť napríklad k špecifikácii autora pravidla, popisu pravidla, prípadne odkazu na dodatočné informácie o danom type malvéru. Sekcia Metadat je znázorňená na obrázku 3.3.

```
rule moje_pravidlo
{
    // Moje pravidlo pre detekciu textoveho alebo
    // hexadecimalneho retazca v suboroch mensich ako 1MB
    meta:
        description = "detekcia retazcov v malych suboroch"
        author = "Tomas Duris"
    strings:
        $textovy_retazec = "retazec"
        $hexadecimalny_retazec = { 00 01 02 03 04 05 }
    condition:
        (
            $textovy_retazec or
            $hexadecimalny_retazec
        ) and
        filesize < 1MB
}
```

Obr. 3.3: Komplexné YARA pravidlo obsahujúce komentáre a sekciu metadat.

3.2 Aktuálne moduly nástroja YARA

Jedným zo spôsobov rozšírenia funkcionality nástroja YARA sú moduly. Pomocou nich je možné definovať vlastné dátové štruktúry alebo funkcie, ktoré sú následne prístupné v YARA pravidlách. Vytvorenie vlastného modulu je pomerne jednoduché a nevyžaduje si podrobnú znalosť interného fungovania samotného nástroja. Využívajú sa sprístupnené API funkcie, ktoré poskytuje YARA. Moduly sú písané v jazyku C, tým pádom si vytvorenie nového modulu vyžaduje aj jeho znalosť. Využitie modulu v YARA pravidlách je podmienené jeho vložením pomocou kľúčového slova `import` nasledovaného názvom modulu. Okrem možnosti vytvorenia modulu sú sprístupnené aj vstavané moduly, vytvorené či už autormi YARA, alebo komunitou okolo tohto nástroja. Vstavané moduly je možné rozdeliť na tie, ktoré neplnia žiadne vlastné štruktúry a len sprístupňujú funkcie v YARA pravidlách. Medzi takéto moduly patrí:

- **Time** - Sprístupňuje len funkciu `now()`, ktorá vracia unixovú časovú pečiatku.
- **Console** - Umožňuje zaznamenávať ladiace a kontrolne informácie počas vykonávania pravidiel. Pokiaľ nie je zvolené inak, informácie sú posielané na štandardný výstup.
- **String** - Zahŕňa funkcie na manipuláciu s textovými reťazcami. Patria sem funkcie na prevod reťazca do číselnej podoby v desiatkovej alebo v inej zvolenej sústave a získanie dĺžky reťazca.
- **Math** - Umožňuje spočítanie hodnôt na základe rôznych častí vstupného súboru. Medzi funkcionalitu pridanú modulom patrí: získanie hodnoti π, spočítanie maximálnej či minimálnej hodnoty dvoch čísel, získanie absolútnej hodnoty, funkcie na výpočet korelácie a mnoho ďalších.

- **Hash** - Slúži na výpočet hodnoty hash funkcií pre zvolené časti súbora. Vypočítané hodnoty je možné využívať v YARA pravidlách a hľadať na základe nich zhodu s očakávanou hodnotou. Medzi aktuálne podporované funkcie patrí: `md5`, `sha1`, `sha256` pre výpočet hashu a `checksum32`, či `crc32` pre výpočet kontrolného súčtu.
- **Magic** - Sprístupňuje možnosť identifikácie súboru. Využíva prvky príkazu `'file'` v unixových systémoch.
- **Cuckoo** - Pridáva možnosť využívať v YARA pravidlách behaviorálne prvky. Informácie o správaní sa súboru pri spustení sú získane z izolovaného prostredia nazývaného sandbox. Obohacuje statickú analýzu o prvky dynamickej analýzy a umožňuje vytváranie pravidiel založených aj na tom, ako sa daný súbor správa a nie len na tom, čo obsahuje.

Druhý typ modulov tvoria moduly pracujúce so špecifickým typom analyzovaného súboru. Tieto moduly sa starajú o rozbor súboru a sprístupnenie jeho častí a špecifík vo vytvorených štruktúrach do YARA pravidiel. Patrí sem **PE**, **ELF**, **LNK**, **Dotnet** a **Mach-O** modul. Každý zo spomenutých modulov sprístupňuje charakteristické vlastnosti, atribúty, položky dostupné v hlavičkách a iné informácie špecifické pre daný typ súboru. Mach-O modul nie je spomenutý v oficiálnej dokumentácie, keďže sa jedná o modul, ktorý stále nebol plne dokončený. Mach-O modulu sa venuje aj časť práce zaoberajúca sa o rozšírenie ekosystému nástroja Yara-X, spomenutého v kapitole 4. Príklad YARA pravidla využívajúceho Mach-O modul a jeho funkcie je ukázaný na obrázku 3.4.

```
import "macho"

rule macho_pravidlo
{
    condition:
        macho.magic == 0xfeedface // Mach-O 32-bit
}
```

Obr. 3.4: YARA pravidlo využívajúce modul macho a jeho funkcie.

3.3 Aktuálne využitie nástroja YARA vo firme Gen Digital

Nástroj YARA je jeden z najviac využívaný prostriedkov vo firme Gen Digital v boji s malvérom. Využíva sa k analýze a detekcii potenciálne škodlivých súborov. Využíva sa pritom rozšírená interná verzia, ktorá obsahuje rôzne modifikácie v samotnom jazyku alebo aj v dostupných moduloch. Nástroj YARA sa nevyužíva len samotne, ale aj ako súčasť množstva nástrojov tvoriacich interný ekosystém, ktorého rozšírenie a vylepšenie je súčasťou diplomovej práce. Medzi príklady spomínaných nástrojov, ktorým sa práca bude v nasledujúcich kapitolách venovať, patrí `Yaramod`[3], `YLS`[4] a `YARI`[5] [17].

[3] https://github.com/avast/yaramod
[4] https://github.com/avast/yls
[5] https://github.com/avast/yari

Yaramod je voľne dostupný nástroj slúžiaci k prevodu YARA pravidla do abstraktného syntaktického stromu. Vďaka tomu je možné pravidlo analyzovať a modifikovať jeho časti bez nutnosti manipulácie s celým pravidlom. V abstraktnom syntaktickom strome sú zachované vzťahy medzi jednotlivými časťami pravidla. Abstraktný syntaktický strom pre pravidlo 3.3 je v textovej podobe vypísaný na obrázku 3.5. Yaramod okrem manipulácie s pravidlami umožňuje aj ich vytváranie pomocou deklaratívneho prístupu z kódu. Dostupné je napojenie na C++ a Python jazyk. Využitý je vlastný parser, ktorý umožňuje oddialiť syntaktickú kontrolu a aj napriek tomu že nevyužíva parsovanie odolné voči chybám dokážeme týmto spôsobom tolerovať niektoré chyby. Aj napriek využitiu vlastného parsera je zachovaná parita s tým, akým spôsobom parsuje pravidlá nástroj YARA. Jedným z ďalších rozšírení, ktoré ponúka nástroj Yaramod je automatické formátovanie pravidiel, umožňujúce zachovanie konzistencie a jednotného štýlu naprieč všetkými pravidlami využívanými firmou Gen Digital. Dôležitým poznatkom je, že Yaramod žiadnym spôsobom nevyhľadáva zhody v pravidlách a vstupnom súbore. Slúži len na spracovanie a manipuláciu s pravidlami. Podporované sú všetky moduly, ktoré sú dostupné v nástroji YARA. Okrem voľne dostupnej verzie nástroja Yaramod existuje aj interná verzia obsahujúca nadstavbu funkcionality a informácii, ktoré nie sú dostupné verejnosti. Práve internú verziou nástroja Yaramod využíva **Y**ara **R**ules **T**ool**C**hain (YRTC), ktorý sa stará o vykonávanie operácii nad internou sadou pravidiel udržiavanou v rámci firmy.

Patria sem operácie ako:

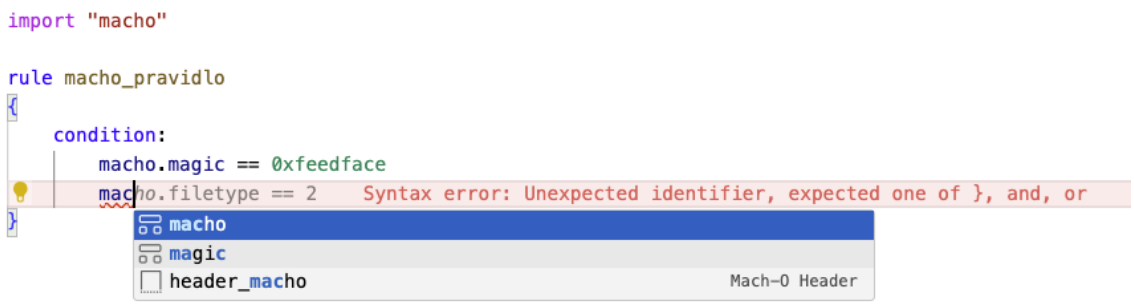
- `auto-format` - Formátovanie YARA pravidiel do jednotného formátu.
- `check-format` - Kontrola či sú pravidlá správne naformátované.
- `compile` - Pokus o kompiláciu YARA pravidiel jedno po druhom. Varovania sú interpretované ako chyby a príkaz zlyhá ak aspoň jedno pravidlo je neskompilovateľné.
- `transform <format>` - Transformovanie pravidiel do špecifikovaného formátu využiteľného iným interným nástrojom alebo inými internými systémami.

```
==== RULE: moje_pravidlo
And[0x100ac9df0]
    Parentheses[0x100acbef0]
        Or[0x100a8ebf0]
            String[0x100adbb30]  id=$textovy_retazec
            String[0x100adbb30]  id=$hexadecimalny_retazec
    Lt[0x100ae3230]
        Filesize[0x100acbb30]
        IntLiteral[0x100aa85f0]  value=100MB
```

Obr. 3.5: Abstraktný syntaktický strom vytvorený za využitia nástroja Yaramod.

Yara **L**anguage **S**erver je program implementujúci LSP (Language Server Protocol). LSP je špecifikácia vyvinutá spoločnosťou Microsoft, ktorá poskytuje možnosť implementovať klienta a server nezávisle od seba [19]. Program YLS je kompatibilný s väčšinou moderných editorov. Poskytuje funkcionalitu ako:

- Automatické doplnenie kódu - Poskytuje užívateľovi možnosť automatického doplnenia písaného pravidla o jednoduché ukážky alebo referencie na iné pravidlá v súbore. Túto funkcionalitu demonštruje obrázok 3.6.
- Zvýrazňovanie chýb a varovaní - YLS poskytuje užívateľovi spätnú väzbu o práve písanom alebo upravovanom pravidle. V editore sa zobrazí správa, ktorá zvýrazňuje miesto s chybou a je obohatená o krátky popis chyby.
- Automatické formátovanie pravidiel - Umožňuje úpravu formátu súboru s pravidlami. Slúži na zachovanie konzistentnosti a jednotného štýlu naprieč všetkými pravidlami. Užívateľ môže spustiť formátovanie pomocou klávesovej skratky alebo nastaviť jeho automatické spustenie pri uložení súboru.
- Navigácia medzi referenciami - YLS dokáže zostrojiť prepojenia medzi symbolmi používanými v pravidlách. Na základe prepojení vieme prechádzať medzi symbolmi a ich definíciami aj naprieč pravidlami.

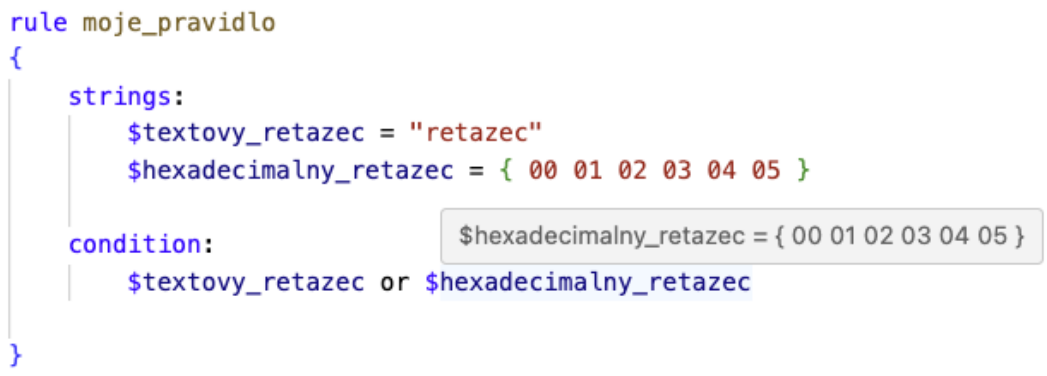


Obr. 3.6: Ukážka rozšírenia zahŕňajúceho YLS a jeho funkcionalitu pre VS Code editor.

Ďalšia funkcionalita zahŕňa napríklad zobrazovanie podrobných informácii o symbole (ak naň prejdeme kurzorom), alebo zobrazenie hierarchie symbolov súboru. Hlavným cieľom programu YSL je poskytnúť efektívnu manipuláciu s YARA pravidlami. YLS využíva nástroj Yaramod k parsovaniu pravidiel a získavaniu informácii o moduloch. Písané pravidlo je syntakticky neúplné po celú dobu až kým sa nedostane do priebežnej finálnej podoby. Ideálny scenár by bol využiť parser odolný voči chybám, ktorý vie v akom približne stave sa parsovanie nachádza a dokáže ponúknuť ďalšie možné cesty, ktorými sa vydať v podobe nápovied pre doplnenie kódu. Bližšiemu popisu parsovaniu odolnému voči chybám sa práca venuje v návrhu a implementačnej časti. Aktuálna implementácia YLS za využitia nástroja Yaramod umožňuje tento spôsob obísť pomocou ukladania si posledného validného objektu z Yaramod parsera. V prípade, že je tento spôsob nedostačujúci, napríklad pri využití funkcionality poskytujúcej automatické dopĺňanie kódu, YLS pracuje priamo so vstupným textom. Využije sa aktuálny pozícia kurzora a Yaramod parser sa úplne obíde.

Jednou z aktuálnych možností ako sa vysporiadať s chybou v YARA pravidle je použiť vstavaný modul `console`, ktorý umožňuje vypísať ladiace informácie o pravidle. Tento prístup si však vyžaduje modifikáciu samotného pravidla.

YARI je nástroj napísaný v jazyku Rust, ktorý umožňuje interaktívne vyhodnocovanie YARA výrazov. YARI je možné použiť ako rozšírenie pre YLS, ako knižnicu pre jazyk Python, alebo ako interaktívne konzolové prostredie. Obrázok 3.7 ukazuje akým spôsobom je možné využiť YARI a zobraziť hodnotu premennej priamo v editore. Vstupom je YARA výraz a na výstupe je vyhodnotenie výrazu. YARI podporuje všetky vstavané moduly, konštanty, štruktúry, ale aj volania funkcií s parametrami. Umožňuje vyhodnocovanie jednoduchých výrazov aj priamo v editore ako rozšírenie nástroja YLS. Pri umiestnení kurzora napríklad nad premennú `hexadecimalny_retazec`, sa zobrazí jej hodnota `{ 00 01 02 03 04 05 }`. Je to hodnota, s ktorou pracuje aj nástroj YARA. Okrem vyhodnocovania jednoduchých výrazov YARI umožňuje prácu s pokročilejšími konštrukciami, ako napríklad binárne operácie, alebo prácu s reťazcami [18].



Obr. 3.7: Rozšírenie YARI, ktoré je súčasťou balíka YLS.

Kapitola 4

Modulárna architektúra YARA-X

YARA pomerne nedávno oslávila svoje pätnáste narodeniny. Od počiatočnej verzie až po dnešnú prešla mnohými úpravami a rozšíreniami. YARA sa začala využívať nie len ako samostatný nástroj, ale aj ako súčasť iných nástrojov, či celých ekosystémov. Prichádzali požiadavky na to, čo má YARA obsahovať a každý si našiel inú cestu, ako ju využívať. To spolu so zvyšujúcou sa neprehľadnosťou kódu a vidinou vytvorenia jednotného ekosystému, ktorý okrem pôvodnej funkcionality zahrnie aj rôzne iné komunitné vylepšenia, viedlo k vytvoreniu nového projektu s názvom YARA-X. Iniciatívu okolo YARA-X vedie jej pôvodný autor Victor M. Alvarez. Pôvodne bol tento projekt braný ako experiment, aby sa zistilo, či prinesie benefity.

YARA-X je vyvíjaná v jazyku Rust, ktorý umožňuje zachovať porovnateľnú rýchlosť s jej pôvodnou verziou napísanou v jazyku C a pritom adresovať veľké množstvo bezpečnostných nedostatkov už len tým, ako samotný jazyk v princípe funguje. Bližšiemu popisu jazyku Rust a jeho výhod, najmä v porovnaní s jazykom C, sa práca ďalej venuje v kapitole 4.1.

Už krátko po vzniknutí iniciatívy okolo nástroja YARA-X sa zistilo, že tento projekt prinesie veľké množstvo benefitov, ako napríklad jeho modularita s čím je spätá aj jednoduchá rozšíriteľnosť či udržateľnosť a čitateľnosť kódu. Pôvodná YARA trpela aj množstvom bezpečnostných chýb spojených s prístupom do pamäte. Aj keď v čase písania tejto práce stále nie je YARA-X oficiálne vydaná, tak už nie je v experimentálnej fáze a očakáva sa, že k jej vydaniu dôjde čoskoro. Victor M. Alvarez, v komunikácii povedal, že nastala fáza, kedy sa pôvodná YARA postupne presúva do fázy údržby a nebude sa pridávať žiadna nová funkcionalita. Všetká nová funkcionalita je smerovaná k projektu YARA-X[1] a má byť jedným z hlavných dôvodov ako presvedčiť ľudí aby na ňu časom prešli.

Vo firme Gen Digital a predtým vo firme Avast využívame nástroj YARA na dennodennej báze. Jednou z hlavných motivácii tejto práce je nadviazať na už vzniknuté úsilie o jej modernizáciu. Radi by sme už od začiatku podporovali tento projekt a boli jedným z jeho hlavných prispievateľov a užívateľov. Táto iniciatíva je za firmu Gen Digital autorom práce a práca si kladie za cieľ rozšíriť už vzniknutý ekosystém o ďalšie nástroje, ktoré uľahčia jej využívanie. Rovnako cieľom je aj integrácia verejných, prípadne aj interných nástrojov firmy Gen Digital do vzniknutého ekosystému. Proces rozširovania ekosystému nástroja YARA-X je pravidelne konzultovaný aj s autorom nástroja, Victorom M. Alvarezom. Cieľ práce spočíva vo vytvorení nového modulu pre parsovanie Mach-O súborov, predloženia knižnice pre jednoduché spracovanie a zobrazenie nadobudnutých informácii, samostatného univerzálneho testovacieho rozhrania a v neposlednom rade implementova-

[1] https://github.com/VirusTotal/yara-x

nia parsera odolného-voči chybám,ktorý umožní využívanie nástroja YARA-X pre potreby využitia v Integrovaných vývojových prostrediach (IDE).

4.1 Jazyk Rust

Použitie jazyka Rust je jednou z najväčších zmien, ktoré projekt YARA-X prináša. Väčšina nízkoúrovňových programov, ktoré majú za cieľ byť rýchle, využíva jazyk C alebo C++. Tieto jazyky síce prešli za posledné roky istou mierou modernizácie, stále však neposkytujú mechanizmus ako zaistiť bezpečnosť pri práci s pamäťou a nechávajú zaistenie bezpečnosti na samotnom autorovi kódu [15]. Aj keď je to dobrá praktika, v praxi to znamená, že veľké množstvo kódu obsahuje zraniteľnosti, ktoré je nutné v budúcnosti riešiť. Bolo dokázané, že až 70% zraniteľností v kóde pochádza zo zlej správy pamäte [21]. O vyriešenie problému so správou pamäte sa pokúša Rust.

Rust je považovaný za stále pomerne nový programovací jazyk. Vznikol ako vedľajší projekt zamestnanca spoločnosti Mozilla v roku 2006. Názov Rust vznikol zo slova *robust*, ktoré malo najlepšie vyjadrovať vlastnosti jazyka. V roku 2010 Rust prerástol rozmery vedľajšieho projektu a začala ho sponzorovať celá spoločnosť Mozilla. O dva roky neskôr bola vydaná prvá verzia jazyka Rust. Aj v dnešnej dobe je Rust stále silno podporovaný a nová stabilná verzia vychádza každých šesť týždňov. Rust sa v poslednej dobe teší veľkej obľube a vznikajú rozšírenia a knižnice pre tento jazyk. Jedna z najpoužívanejších knižníc sa nazýva rust-analyzer[2]. Jedná sa o knižnicu pre sémantickú analýzu kódu napísaného v jazyku Rust. Táto knižnica sa používa ako súčasť servera implementujúceho LSP. Podobne ako YLS pre jazyk YARA, rust-analyzer umožňuje integráciu sémantickej kontroly, automatického doplňovania kódu, zvýrazňovania chýb v kóde, či navigáciu medzi referenciami naprieč zdrojovými súbormi pre väčšinu používaných editorov zdrojového kódu. Funguje ako rozšírenie do väčšiny moderných editorov. Rust je už ôsmy rok po sebe vyhodnotený ako najobľúbenejší programovací jazyk medzi vývojármi na známej platforme StackOverflow[3]. Podľa [16], medzi hlavné dôvody, prečo si vývojári obľúbili tento jazyk patrí:

- Správa pamäte (bezpečnosť) - Rust využíva princíp *'ownership and borrowing'*. Princíp vlastníctva implementuje striktné pravidlá, ktoré sa kontrolujú už pri kompilácii a zabraňujú častým chybám pri využívaní a pristupovaní do pamäte. Tie by v iných jazykoch mohli byť zistené až za behu a mohli by viesť k nedefinovanému správaniu alebo k neočakávanému zlyhaniu programu. Fungovanie je založené na báze, ktorá spočíva v tom, že každá hodnota v Ruste má práve jedného vlastníka, ktorý je zodpovedný aj za jej uvoľnenie z pamäte, keď sa dostane mimo prostredia platnosti. Vlastník hodnoty *'owner'* je jediný, kto môže požičať *'borrow'* referenciu na túto hodnotu iným častiam kódu, ktoré ju môžu používať bez toho, aby prebrali jej vlastníctvo. Je možné preniesť vlastníctvo hodnoty aj na inú premennú, vtedy však pôvodný vlastník stráca platnosť preto, aby sa predišlo neplatným referenciám. Rust rovnako využíva tieto pravidlá a rieši pomocou nich problémy aj pri súbežnom prístupe k zdieľaným zdrojom.

- Cargo - nástroj Cargo, ktorý Rust ponúka je jedným z hlavných dôvodov, prečo si ľudia tento jazyk obľúbili. Umožňuje napríklad jednoduchú správu softvérových závislostí, pridávanie a odoberanie závislostí a kompiláciu programu do verzie vhodnej na distribúciu. Pomáha k vytvoreniu jednoduchého a efektívneho prostredia pre prácu.

[2] https://rust-analyzer.github.io
[3] https://survey.stackoverflow.co/2023/

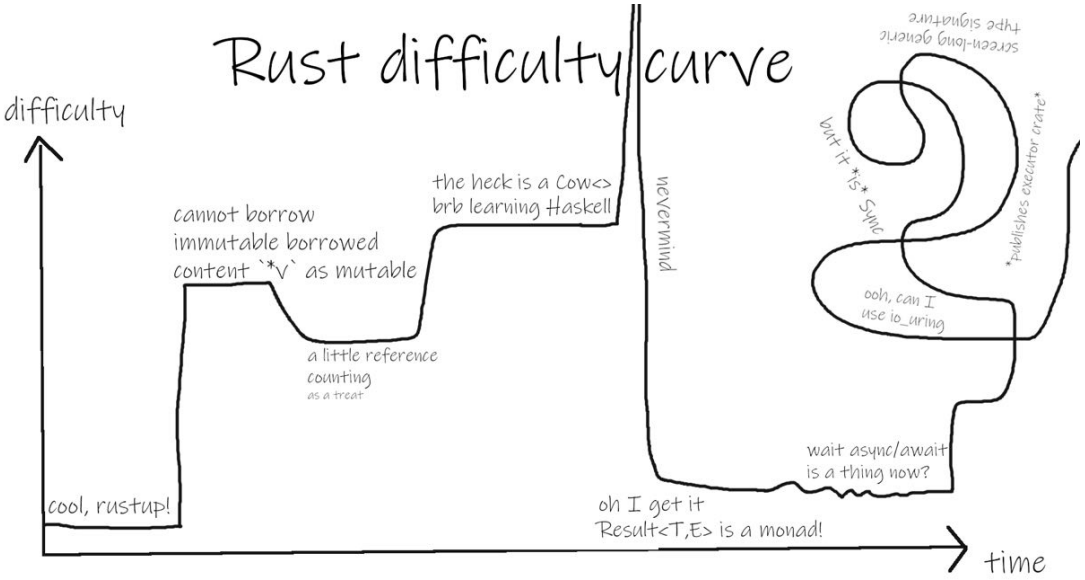
- Pattern matching - Podobne ako Haskell, aj Rust prináša možnosť efektívneho hľadania zhody medzi potenciálne komplexnou dátovou štruktúrou a definovaným vzorom. Umožňuje programátorom ošetriť rôzne možné scenáre, podobne ako pri if-else konštrukcii, a pritom zachovať čitateľnosť, čistotu kódu a ošetriť všetky potenciálne možnosti.

- Automatická detekcia typov - Rust umožňuje automaticky detekovať typy pri kompilácii programu na základe kontextu. Znamená to, že vo väčšine prípadov nie je potrebné vopred špecifikovať typ premennej alebo parametru funkcie a kompilátor je schopný si to sám zistiť pri kompilácii.

- Rýchlosť - Rýchlosť jazyka Rust je na podobnej úrovni ako pri jazyku C a je radený medzi rýchle programovacie jazyky. Vývojári majú veľkú kontrolu nad správou pamäte a optimalizáciami pomocou princípu *'ownership and borrowing'*. Väčšina kontrol prebieha pri kompilácii, ktorá je síce pomalšia, ale benefituje z toho rýchlosť behu programu. Rust nepoužíva *garbage collector* a takmer všetky úrovne abstrakcie, ktoré poskytuje nepridávajú žiadnu dodatočnú réžiu počas behu programu.

Syntax jazyka Rust pripomína syntax jazyka C, respektíve C++. Jedná sa o pomerne komplexnú syntax, ktorá cieli na poskytnutie nízko-úrovňovej kontroly nad hardvérom pre programátora a zároveň na zachovanie rýchlosti a bezpečnosti kódu. Rust poskytuje viacero prvkov, ktoré sa o to pričiňujú. Medzi ne patria napríklad makrá, parametre životnosti premenných, pattern matching, uzávery alebo rozhrania [7]. Na rozdiel od jazyka C++ v jazyku Rust neexistuje koncept tried. Namiesto toho Rust používa štruktúry pre definovanie dát a implementácie pre definovanie metód nad týmito štruktúrami.

Rust je považovaný za pomerne zložitý jazyk pre začiatočníkov a vyžaduje si väčšie množstvo času na pochopenie jeho princípov a ich správneho využitia. Avšak po osvojení si princípov Rustu a správneho využívania jeho vlastností výrazne stúpa efektivita programátora pri písaní čistého, efektívneho, rýchleho a predovšetkým bezpečného kódu. Rust podobne ako napríklad aj Haskell má strmšiu učiacu krivku, keďže si vyžaduje prvotné pochopenie pokročilejších konceptov, ktorými dané jazyky disponujú a nie sú bežné pri iných jazykoch. Rovnako, keďže Rust je pomerne nový programovací jazyk, počet počet materiálov a zdrojov k jeho naučeniu nie je taký rozsiahly ako pri iných programovacích jazykov. Má to takisto vplyv aj na učiacu krivku, ktorej strmosť je výraznejšia ako napríklad pri jednoduchších jazykoch, ktoré sú tu už dlhšiu dobu. Porovnanie učiacej krivky pre Rust a iné programovacie jazyky je vyobrazené na obrázku 4.1. Autor popisuje priebeh učenia jazyka a nutnosť zvládnutia jeho pokročilých konceptov, ktoré ho častokrát ešte viac zamotali a jej priebeh má teda skôr zábavný charakter.



Obr. 4.1: Názorná ukážka učiacej krivky pre Rust. Prevzaté z [6].

4.2 Architektúra nástroja YARA-X

Jedným z hlavných benefitov nástroja YARA-X je jeho modulárna architektúra [10]. Je to základný stavebný kameň, na ktorom Victor M. Alvarez postavil ekosystém YARA-X. Cieľom tejto práce je ho ďalej rozšíriť. YARA-X využíva *cargo workspace* prostredie, ktoré umožňuje jednotnú správu viacerých balíkov alebo knižníc v rámci jedného projektu. Oproti pôvodnému nástroju YARA nie je všetká funkcionalita obsiahnutá v jednej knižnici. Prostredie YARA-X extrahuje časti ako parser, alebo makrá do samostatných balíkov s ich vlastnými závislosťami, ktoré je možné samostatne využívať. Medzi balíky aktuálne obsiahnuté v prostredí YARA-X patrí:

cli: Obsahuje funkcionalitu pre využívanie nástroja YARA-X pomocou príkazového riadku. Stará sa o spracovanie vstupných argumentov, volanie a prepojenie príslušných balíkov nástroja YARA-X alebo o vykreslenie nápovedy užívateľovi. Umožňuje rekurzívne prechádzať adresáre alebo súbory poskytnuté na vstupe a spúšťať nad nimi zvolené funkcie, ktoré poskytuje YARA-X.

fmt: Formátovací nástroj pre jazyk YARA. Jedná sa o funkcionalitu, ktorú vo firme Gen Digital poskytoval nástroj Yaramod, keďže pôvodná YARA túto možnosť neobsahovala. Ide o automatické formátovanie YARA pravidiel. Umožňuje udržiavať jednotný štýl naprieč všetkými YARA pravidlami. V aktuálnej verzii tento nástroj nie je konfigurovateľný a je teda možné využívať len jeden preddefinovaný štýl.

macros: Poskytuje sadu makro výrazov, ktoré je možné používať v kóde. Umožňuje definíciu `main` funkcií pre moduly nástroja YARA-X, zjednodušenie práce chybovými stavmi a generovanie príslušných chybových správ. Ďalšia z funkcionalít, ktorú poskytuje je možnosť definovať určité funkcie v YARA-X moduloch ako funkcie, ktoré je možné volať priamo z YARA pravidiel pomocou `module_export` makra. Rovnaká funkcionalita bola implemen-

tovaná aj v pôvodnom nástroji YARA. Poskytuje to možnosť pridať dodatočnú funkcionalitu a výpočetnú schopnosť pre jazyk YARA. Pre možnosť volať vybrané funkcie aj z WASM (*WebAssembly*[4]) kódu bolo pridané makro `wasm_export`. Využitiu WebAssembly v nástroji YARA-X sa bude práca bližšie venovať na konci tejto sekcie.

parser: Pôvodne bolo parsovanie jazyka YARA a s ním spojená syntaktická a sémantická analýza vykonávaná spoločne s ostatnou funkcionalitu nástroja YARA v knižnici `libyara`. Toto viedlo k úzkemu prepojeniu jednotlivých častí a nebolo možné ich jednoducho oddeliť. YARA-X používa parser, ktorý je umiestený v samostatnom balíku a nemá žiadne závislosti na `yara-x` knižnicu. Pôvodne sa využíval ručne vytvorený parser spolu s generátorom pre lexikálnu analýzu a gramatiku. Na generovanie kódu pre lexikálnu analýzu sa využíval nástroj `flex`[5] a na generovanie gramatických pravidiel nástroj `bison`[6]. YARA-X využíva nástroj `Pest`[7]. Pest využíva samostatne oddelenú gramatiku, ktorá sa nemieša so samotným kódom. Gramatika je založená na výrazoch `PEG` (**P**arsing **E**xpression **G**rammar), ktoré pripomínajú regulárne výrazy. Lexikálna analýza je vykonávaná pomocou pravidiel definovaných v gramatike. Pravidla sa skladajú z alfanumerických znakov, voliteľných výrazov a napríklad znakov pre opakovanie, rozsah, alebo alternatívy.

Pravidlo pre identifikátor by mohlo vyzerať napríklad takto:

```
alpha = { 'a'..'z' | 'A'..'Z' }
digit = { '0'..'9' }

ident = { ( alpha | digit) + }
```

`Alpha` a `digit` sú ďalšie pravidlá definujúce písmená a čísla, `|` je operátor alternatívy, alebo aj logický výraz `OR` a `+` vyjadruje možnosť opakovania daného prvku 1 alebo viackrát.

Výstupom nástroja `Pest` je dvojica, pozostávajúca z časti vstupu a príslušného pravidla, s ktorým nastala zhoda. Neumožňuje vytvorenie abstraktného syntaktického stromu a táto štruktúra je vytvorená samostatne na základe poskytnutých párov vstupného textu a pravidiel. Jednou z výhod YARA-X je možnosť následného zobrazenia abstraktného syntaktického stromu pomocou spustenia nástroja s argumentami `debug` a `ast`. Použitie nástroja `Pest` alebo všeobecne parser generátora prináša viacero výhod, akými sú napríklad jednoduchosť, čitateľnosť a zaručená správnosť pre danú gramatiku. Ich funkcionalita je však často obmedzená, nepodporujú parsovanie odolné voči chybám a riešenie niektorých hraničných situácii nemusí byť vždy podporované. Táto práca predkladá vylepšenia aktuálneho parsera umožňujúce napríklad flexibilnejšie parsovanie, alebo odolnosť voči chybám. Vďaka adresovaniu týchto nedostatkov je možné ďalej ekosystém nástroja YARA-X rozširovať.

proto: Obsahuje definíciu protobuf[8] súboru pre YARA-X moduly. Pôvodná YARA využívala statické štruktúry k uloženiu informácii získaných z parsovania rôznych typov súborov pomocou vstavaných modulov (viz. 3.2). YARA-X využíva protobuf formát k uloženiu a serializácii spomenutých informácii. Okrem základnej štruktúry sú definované aj rôzne metadáta, ako napríklad meno modulu, alebo nastavenia jednotlivých polí a štruktúr. V YARA pravidlách následne pomocou `import <module>` je možné sprístupniť a využívať všetky uložené informácie. Protobuf štruktúra je oddelená a nezávislá od samotného kódu a narozdiel od pôvodných statických štruktúr, je možné tieto informácie jednoducho spracovať a využívať aj v rôznych iných, či už vstavaných alebo externých nástrojoch. Práca

[4] https://webassembly.org/
[5] https://github.com/westes/flex
[6] https://www.gnu.org/software/bison/
[7] https://pest.rs
[8] https://protobuf.dev

predkladá využitie tejto štruktúry k jednoduchému zobrazeniu výstupu naparsovaných informácií z príslušného súbora v strojovo čitateľnej a aj vizuálne prívetivej podobe [10].

py, **capi** a **go**: Jedná sa o implementácie napojení funkcionality YARA-X na jazyky Python, C a Go. Poskytujú prístupové rozhrania a množstvo funkcií, ktoré je možné volať priamo z kódu napísaného v spomínaných jazykoch. Takisto umožňujú aj kompiláciu YARA pravidiel a skenovanie jedného, alebo viacerých súborov pomocou YARA pravidiel.

lib: Jadro ekosystému YARA-X sa nachádza v balíku s názvom `lib`. Obsahuje zdrojové kódy pre väčšinu funkcionality, ktorú YARA-X poskytuje. Umožňuje kompiláciu YARA pravidiel, konkrétne ich reprezentáciu vo forme abstraktného syntaktického stromu do medzipodoby (***H**igh-level **I**ntermediate **R**epresentation)*, ktorá je využitá ku generovaniu WASM kódu. WASM je skratka pre *WebAssembly*, jedná sa o binárny formát inštrukcií pre zásobníkový virtuálny stroj. Pôvodná YARA využívala vlastný trojadresný kód. Využitie WASM umožňuje nezávislosť na cieľovej platforme a potenciálne spúšťanie nástroja YARA vo webových prehliadačoch. Okrem toho sú jeho výhodami aj efektívnosť a rýchlosť na širokej škále zariadení. Je bezpečný a obsahuje pamäťovo bezpečné a izolované prostredie k spúšťaniu kódu. V neposlednom rade poskytuje textovú reprezentáciu, ktorá je jednoducho laditeľná a testovateľná.

Balík `lib` obsahuje aj definíciu vstavaných modulov 3.2 rozširujúcich funkcionalitu nástroja YARA-X. Jedná sa o rovnakú sadu modulov ako pri pôvodnom nástroji. Jediná väčšia funkcionálna zmena predstavuje využívanie už vyššie spomenutých protobuf štruktúr k ukladaniu informácii získaných z modulov. Každý modul okrem zdrojového kódu obsahuje aj protobuf štruktúru definujúcu aké informácie sa očakávajú na výstupe parsovania. Tento súbor sa postupne plní získanými informáciami a je sprístupnený v YARA pravidlách.

4.3 Možné vylepšenia

Architektúra, ktorú YARA-X prináša je jednoducho rozšíriteľná a predstavuje dobrý základ pre využívanie YARA-X, nie ako jedného nástroja, ktorý rieši jeden konkrétny problém, ale ako sadu širokého spektra nástrojov, ktorá umožňuje a cieli na vykonávanie väčšiny potrieb malvérového analytika. Už teraz YARA-X priniesla viacero benefitov oproti pôvodnej implementácii, ako napríklad vstavané formátovanie pravidiel, jednoduché vkladanie knižníc do ekosystému, ukladanie informácii získaných z parsovania súborov v serializovanej podobe či využitie univerzálnej WASM reprezentácie a tým pádom aj možnosť integrovanie nástroja YARA do webových prehliadačov.

Táto diplomová práca nadviaže na spomínané vylepšenia a prinesie možnosť jednoduchého získania a zobrazenia informácii z modulov, implementáciu prototypu parsovania odolného voči chybám, ktoré umožní ďalej rozširovať ekosystém o nástroje ako `YLS`, `YARI`, alebo množstvo iných nástrojov, ktoré nebudú musieť využívať ich vlastný parsera. Okrem integrovania už existujúcich nástrojov práca umožní jednoduché vytváranie nových nástrojov, ktoré budú môcť byť vložené priamo do YARA-X ekosystému. Aktuálne nástroj YARA-X využíva benefity jazyku Rust, s čím je spojená čistota kódu a jednoduché testovanie priamo z kódu alebo dokumentácie. Väčšina funkcií v zdrojovom kóde obsahuje k ním príslušné jednotkové testy. Práca okrem vytvorenia modulu pre jeden zo spustiteľných formátov (konkrétne Mach-O formát) prinesie takisto rozšírenie a zlepšenie pokrytia testov pomocou *fuzzy* testovania či vytvorenia testovacieho rozhrania, ktorý jednotným spôsobom otestuje správnosť implementovaných modulov a umožní jednoduché testovanie budúcich modulov jednotným spôsobom [10].

Kapitola 5

Návrh

Cieľom diplomovej práce je rozšírenie ekosystému nástroja YARA-X. Práca sa zameriava na vylepšenie parsovania YARA pravidiel pomocou statickej analýzy spustiteľných súborov, konkrétne formátu typu Mach-O. Ďalším predstaveným rozšírením je nástroj, ktorý umožňuje užívateľsky prívetivé a strojovo spracovateľné vyobrazenie získaných informácii z modulov pre spracovanie rôznych formátov súborov. Tento nástroj si kladie za cieľ nahradiť iné nástroje pre extrakciu dát a zjednotenie informácii, ktoré sú využívané analytikmi pri písaní YARA pravidiel. Nástroj pre extrakciu dát zo súboru sa využíva aj pri implementácii testovacieho rozhrania pre jednotlivé YARA-X moduly. Posledným predstaveným rozšírením je vytvorenie nového parsera vykonávajúceho syntaktickú analýzu odolnú voči chybám. Cieľom je nahradiť aktuálny parser nástroja YARA-X a rozšíriť jeho funkcionalitu o možnosť získať abstraktný syntaktický strom pre každý, aj nevalidný, vstup spolu s množinou chýb, ktoré sa na danom vstupe vyskytli.

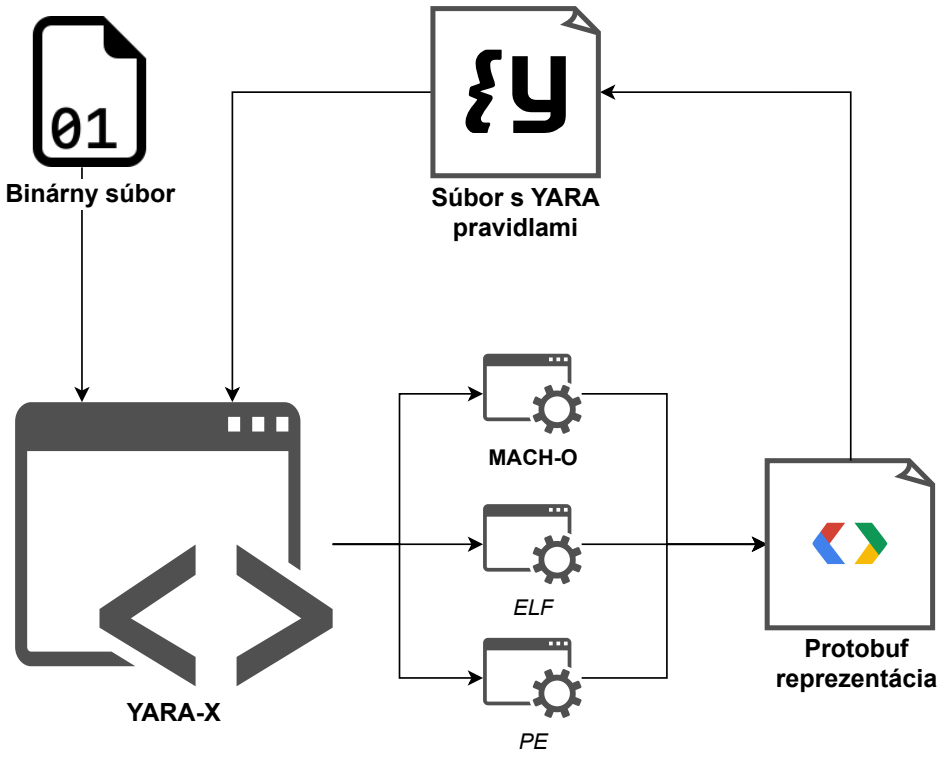
Práca využíva na implementáciu spomínaných rozšírení jazyk Rust, ktorý je zvolený na základe jeho benefitov spojených prevažne s rýchlosťou a výslednou pamäťovou bezpečnosťou kódu. Rovnako nástroj YARA-X, ktorý práca rozširuje je písaný v jazyku Rust a jedná sa o plynulé nadviazanie na už existujúcu architektúru. Navrhnuté riešenie kladie dôraz na spätnú kompatibilitu s pôvodným nástrojom YARA napísaným v jazyku C a rovnako na jednoduchú integráciu do projektu YARA-X. Architektúra riešenia, rovnako ako aj jej implementácia, je komunikovaná aj priamo s autorom projektu YARA, Victorom M. Alvarezom. Projekt YARA-X je v dobe písania diplomovej práce v štádiu finalizácie a prvé produkčné vydanie, ku ktorému prispeje aj táto práca sa očakáva v polovici roka 2024. Projekt YARA-X je však možné už teraz testovať a využívať v jeho experimentálnej podobe.

5.1 Modul pre spracovanie formátu Mach-O

Moduly pre spracovanie súborov v nástroji YARA definujú štruktúru, ktorá je následne naplnená informáciami zo spracovaného súboru. Tieto informácie je možné následne jednoducho využívať v YARA pravidlách. Jedná sa v podstate o pomenovanie jednotlivých častí súbora, jeho rozdelenie do štruktúr a prístup k jednotlivým častiam pomocou ich mena alebo nad nimi definovanými funkciami. Tým je pre užívateľov uľahčené písanie YARA pravidiel, keďže bez prítomností modulov by súbor museli prehľadávať bajt po bajte a rovnako k jednotlivým častiam aj pristupovať. Ako je spomenuté v časti 4.2, YARA-X predstavuje nový spôsob ukladania naparsovaných informácii pomocou modulov [1]. Využíva sa proto-

buf reprezentácia štruktúry súboru, ktorá obsahuje definície jednotlivých častí súboru, ku ktorým je možné z YARA pravidiel pristupovať. O naplnenie protobuf štruktúry sa stará samotný modul. YARA-X obsahuje kostru pre definíciu modulov a práve Mach-O modul je prvým modulom pre spracovanie súborov, ktorý ju využije. Formát Mach-O je určený primárne pre systémy založené na Mach kerneli, ako napríklad MacOS alebo iOS.

YARA-X poskytuje rozhranie pre vývoj modulov a definíciu makro výrazov ako napríklad `module_main` alebo `module_export`. Tieto funkcie po rade definujú vstupný bod modulu a exportovanú funkciu, ktorá je definovaná imperatívnym spôsobom. Predstavuje to rozšírenie funkcionality, ktoré jazyk YARA nepodporuje a je možné ju využívať aj v deklaratívnom prostredí YARA pravidiel. Obrázok 5.1 zobrazuje architektúru využitia modulov a protobuf súborov v YARA pravidlách. Na overenie získaných informácii o súbore pomocou Mach-O modulu je využitý nástroj `otool`[1], keďže sa jedná o referenčný nástroj pre zobrazenie formátu Mach-O. Detailný popis Mach-O modulu spolu s popisom formátu Mach-O je v sekcii 6.



Obr. 5.1: Využitie modulov v ekosystéme YARA-X.

5.2 YARA-X ako nástroj pre extrakciu dát

Nástroj YARA-X, ale aj jeho pôvodná verzia YARA dokáže pracovať s rôznymi formátmi súborov. Výhodou je práve využitie jej modulov pre definíciu štruktúry, jej naplnenie a následné použitie týchto informácii v YARA pravidlách. Množstvo analytikov alebo užívateľov

[1] https://www.unix.com/man-page/osx/1/otool/

ale využíva rôzne iné externé nástroje, ako napríklad už spomínaný `otool` pre zobrazenie častí Mach-O súborov, prípadne nástroj `Retdec`[2], ktorý podporuje veľké množstvo formátov ako ELF, PE, alebo už spomínaný Mach-O.

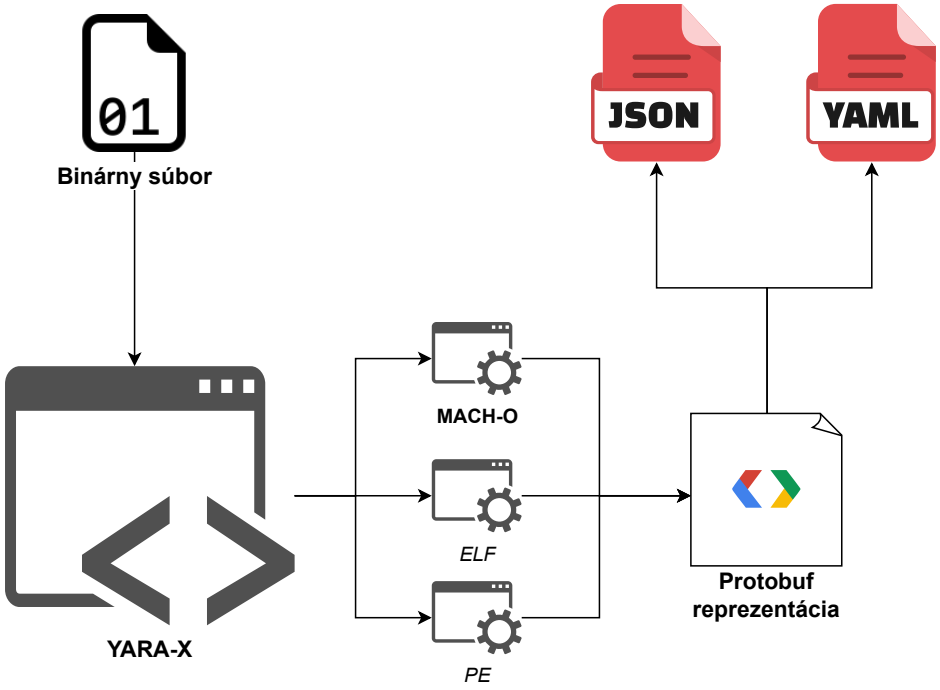
Nástroj Retdec vyvinutý spoločnosťou Avast, je využívaný množstvom analytikov pre prvotnú analýzu súboru a získanie informácii, ktoré sú neskôr využívané pri písaní YARA pravidiel. Jednou z nevýhod tohto prístupu je napríklad to, že Retdec a YARA získané informácie nejakým spôsobom uložia a využijú alebo zobrazia. Tieto informácie avšak nie sú vždy totožné. Aj keď vo väčšine prípadov sa jedná o rovnaké informácie, existuje malá podmnožina súborov, prípadne ich častí, kedy je možné pozorovať nekompatibilitu týchto nástrojov. Výskyt rozdielnych informácii je napríklad pri analýze poškodeného, nečitateľného, či nekompletného súboru. Práve tieto krajné prípady môžu byť rozdielnymi nástrojmi vyhodnotené iným spôsobom. Naskytuje sa teda otázka prečo nevyužiť nástroj YARA, ktorý už podporované formáty súborov analyzuje a parsuje, aj k zobrazeniu týchto informácii. Prečo neumožniť analytikom využitie nástroja YARA nie len na písanie a vyhodnocovanie YARA pravidiel, ale aj ako nástroja na analýzu súboru a poskytovanie informácii o nich. Druhé predstavené vylepšenie rozširuje nástroj YARA o túto funkcionalitu a poskytuje ďalší pomyselný krok k jeho využívaniu nie len ako nástroja na jednu činnosť, ale aj ako skupiny nástrojov poskytujúcich väčšiu funkcionalitu a flexibilitu pri analýze a boji s malvérom.

Vyvinutý nástroj pre extrakciu dát si kladie za cieľ využiť získané informácie z modulov, ktoré sú uložené v protobuf štruktúre. Jedná sa o samostatný nástroj, ktorý ale je súčasťou repozitára a celého ekosystému. Na vstupe sú informácie obsiahnuté v protobuf štruktúre a tieto dáta sú transformované do strojovo-čitateľnej a užívateľsky vizuálne prívetivej podoby. Podporované sú dva najviac využívané formáty v tejto oblasti: **JSON**[3] a **YAML**[4]. Oba formáty podporujú aj farebné zobrazenie a rozlíšenie jednotlivých častí. Formát YAML je naviac rozšírený o komentáre popisujúce jednotlivé sekcie alebo aj o podporu rôzneho zobrazenia decimálnych hodnôt. V množstve prípadov má decimálna hodnota menšiu výpovednú silu, ako napríklad hexadecimálna hodnota, či iný typ jej zobrazenia. Za spomenutie stojí napríklad hodnota rôznych príznakov, ktorá nadobúda hodnoty **1** alebo **0** pre každý bit a teda jej decimálne zobrazenie je len nič-nehovoriace veľké číslo. Ďalším z prípadov, kde toto dáva zmysel je časové razítko UNIX. Jedná sa o počet sekúnd od 1.1.1970 a vo výsledku sa opäť jedná o veľkú nič-nehovoriacu decimálnu hodnotu. Práve z tohto dôvodu je podporovaných viacero formátov zobrazenia decimálnych hodnôt, konkrétne hexadecimálny formát a ľudsky čitateľný formát časového razítka v podobe `2024-01-28 15:53:00 UTC`. Z hľadiska zachovania maximálnej kompatibility a dodržania strojovo-čitateľného formátu, hexadecimálne hodnoty môžu priamo nahradiť tie decimálne. Pri časových značkách sa vloží hodnota ako komentár za pôvodnú decimálnu hodnotu, keďže ju nejde priamo nahradiť bez toho, aby nebol zmenený jej typ z číselnej hodnoty na textový reťazec. Nástroj je integrovaný priamo do projektu YARA-X a je spustiteľný priamo z jeho prostredia pomocou príkazu `yr dump <argumenty> <súbor>`.

Súčasťou nástroja je možnosť špecifikovať, ktorý formát výstupu je žiadaný, alebo aj to, ktorý modul sa má použiť pre naparsovanie a získanie informácii o súbore na vstupe. Okrem manuálnej špecifikácii modulu tento nástroj disponuje aj automatickou selekciou modulu, ktorý môže užívateľ využiť napríklad keď nepozná formát súboru na vstupe a chce jeho detekciu nechať priamo na samotnom nástroji. Obrázok 5.2 popisuje architektúru YARA-X obohatenú o nástroj pre extrakciu dát. Je možné pozorovať ako tento nástroj nadväzuje na

[2] https://github.com/avast/retdec
[3] https://www.json.org/json-en.html
[4] https://yaml.org

prvé predstavené rozšírenie. Modul pre parsovanie Mach-O formátu ukladá nadobudnuté informácie do protobuf štruktúry, ktorá je následne využitá na vstupe nástroja pre extrakciu dát a výsledné informácie sú v užívateľsky prívetivej forme vyobrazené na výstupe. Nástroj sa takisto využíva priamo pri testovaní modulov, kedy pre každý súbor z testovacej sady získa výslednú štruktúru v podporovanom formáte a porovná ju s očakávanou výstupnou štruktúrou. Testovanie je bližšie popísané v kapitole 7.



Obr. 5.2: Architektúra rozšírenia pre extrakciu dát zo súboru.

5.3 Parsovanie odolné voči chybám

Posledné rozšírenie, ktoré je v rámci diplomovej práce predstavené je vytvorenie a využitie parsera odolného voči chybám. Jedným z hlavných cieľov jeho vytvorenia je uľahčiť prácu analytikom pri písaní nových pravidiel pomocou využitia funkcií integrovaných vývojových prostredí. V minulosti nebola odolnosť voči chybám až taká dôležitá kvôli tomu, že sa robil prevažne preklad zdrojového kódu. V dnešnej dobe využívania integrovaných vývojových prostredí však predstavuje významný faktor, ktorý ovplyvňuje rýchlosť a kvalitu písaných pravidiel. Počas písania pravidla alebo ľubovoľného kódu sa tento zdrojový text neustále mení a vo väčšine času je syntakticky nesprávny. Moderný parser má ponúkať aj funkcionalitu akou je napríklad zotavenie sa z chýb, zachovávanie bielych znakov a komentárov, či konštrukcia syntaktického stromu, ktorý priamo zodpovedá zdrojovému textu a mapuje uzly stromu na lokáciu v texte.

Hlavný princíp predstaveného parsera spočíva v tom, že na rozdiel od klasického parsera neskončí pri prvom výskyte chyby. Túto chybu zaznamená spolu s miestom jej výskytu a pokračuje ďalej v texte a pokúsi sa spracovať celý vstup. Výstupom parsera, ktorý je

odolný voči chybám teda môže byť aj nekompletný syntaktický strom, prípadne syntaktický strom obsahujúci uzly, ktoré chybu reprezentujú. V aktuálnej implementácii YARA-X využíva parser generátor, teda nástroj, ktorý vygeneruje kód parsera pomocou jeho textovej definície. Konkrétne sa využíva nástroj Pest, ktorý sa táto práca pokúša nahradiť.

Využitie vlastného parsera so sebou prináša určité výhody aj nevýhody. Medzi jednu z jeho najväčších výhod patrí väčšia kontrola nad parsovaním a ošetrovaním okrajových podmienok jazyka. S tým súvisí aj spomínaná odolnosť voči chybám. Súčasné riešenia parser generátorov nepodporujú odolnosť voči chybám, alebo ňou nedisponujú na dostatočnej úrovni. Motiváciou je takisto vytvorenie jedného univerzálneho parsera, ktorý môže byť používaný nie len vo vnútri projektu YARA-X, ale aj naprieč širokým spektrom iných externých nástrojov. V súčasnosti sa (nie) len vo firme Gen Digital, využíva množstvo nástrojov, ktorým aktuálny parser nevyhovuje práve z dôvodu jeho chýbajúcej funkcionality v podobe odolnosti voči chybám. Tieto nástroje využívajú buď vlastný parser, alebo iný parser, ktorý túto funkcionalitu aspoň v minimálnej forme poskytuje. Jedným z týchto nástrojov je aj spomínaný Yara Language Server. Práve využitie parsera odolného voči chybám by prinieslo jednoduchú integráciu jazykového servera do ekosystému YARA-X, ktorý umožňuje podporu prvkov ako zvýrazňovanie zdrojového kódu v editore, automatické dopĺňanie textu, interaktívne vyhodnocovanie pravidiel pomocou YARI a mnoho iných. Na druhej strane využitie vlastného ručne-písaného parsera so sebou prináša horšiu čitateľnosť kódu, udržovateľnosť kódu a jeho náročnejšie rozširovanie do budúcna.

5.3.1 Vlastnosti parsera

Dôležitým aspektom pri vytváraní ručne-písaného parsera je ujasnenie si požiadaviek, ktoré na neho budú kladené. Výstupom predstaveného parsera je syntaktický strom s nasledujúcimi vlastnosťami:

Biele znaky a komentáre sú súčasťou syntaktického stromu: Nástroje ako YLS, alebo modul `fmt`, ktorý je súčasťou nástroja YARA-X potrebujú reprezentáciu vstupného textu v pôvodnej podobe, ktorá obsahuje aj biele znaky a komentáre. Umožňuje im to priamo pracovať so zdrojovým textom a modifikovať ho. Komentáre sú priamo priradené uzlom, pri ktorých to dáva zmysel, alebo im nadradeným uzlom. Nasledujúci príklad reprezentuje jednoduché YARA pravidlo obsahujúce komentáre na viacerých úrovniach.

```
//Global comment

//Rule comment
rule test
{
    //Rule block comment

    //Strings comment
    strings:
        $a = "foo"
    condition:
        $a
}
```

Je možné si všimnúť viacero komentárov, pričom všetky sú súčasťou syntaktického stromu. Prvý komentár `Global comment` je intuitívne viazaný k zdrojovému súboru a je rov-

nakým spôsobom aj reprezentovaný v syntaktickom strome. Nasleduje `Rule comment`, tento komentár sa síce vyskytuje na podobnom mieste ako `Global comment`, ale je od neho oddelený prázdnym riadkom a prilepený k pravidlu pod ním. Tento komentár je v syntaktickom strome pod uzlom reprezentujúcim pravidlo `test`. Podobným spôsobom sú reprezentované aj komentáre v tele pravidla. `Rule block comment` je priradený uzlu, ktorý reprezentuje telo pravidla a `Strings comment` uzlu vnorenej sekcie `strings`.

Schopnosť reprezentácie chybových stavov: Syntaktický strom obsahuje uzly reprezentujúce chyby v zdrojovom texte. Tieto uzly sú vytvorené pokiaľ došlo ku chybe pri parsovaní vstupu, ale podarilo sa z tejto chyby zotaviť. Dobrým príkladom je YARA pravidlo, ktoré obsahuje chybu v kľúčovom slove `rule`.

```
rulr test
{
    strings:
        $a = "foo"
    condition:
        $a
}
```

Vytvorený syntaktický strom obsahuje uzol reprezentujúci chybový stav. Nasleduje telo pravidla, ktoré je syntakticky korektné a je požadované, aby ho predstavený parser korektne reprezentoval. Toto telo je priradené chybovému uzlu ako jeho potomok. Následný prechod zdrojovým textom a parsovanie prebieha rovnakým spôsobom ako keby ku chybe nedošlo. Okrem vytvorenia takéhoto uzlu dochádza aj k uloženiu miesta výskytu chyby spolu s chybovou hláškou. Niektoré chyby však nedáva zmysel reprezentovať ako chybový uzol, či už z dôvodu, že neobsahujú žiadnych potomkov, alebo sa nepodarilo zotaviť z chyby. Pri výskyte takéhoto typu chyby sa ukladá len miesto výskytu chyby a chybová hláška a v syntaktickom strome to nie je reflektované.

Jednoduchá možnosť aktualizácie stromov a navigácia v ňom: Okrem jednoduchej možnosti navigácie na potomkov uzlov je požadovaná aj jednoduchá navigácia na ich predchodcov. Tento koncept umožňuje definovať rôzne metódy prístupu k syntaktickému stromu, ktoré výrazne uľahčujú prácu s ním. Príkladom môže byť prístup k určitému uzlu na základe jeho umiestnenia v zdrojovom texte. Na základe vedomosti o mieste, kde sa chyba vyskytla, je možné získať aj uzol v syntaktickom strome, ktorý toto miesto pokrýva. Syntaktické spracovanie neobsahuje žiadne prvky sémantickej kontroly. Tá sa očakáva mimo samotného parsera, pri prechode výstupného syntaktického stromu a jeho integrácii s ďalšími časťami. Táto funkcionalita je ponechaná na nástroj YARA-X.

5.3.2 Využitie 'Red-Green' stromov

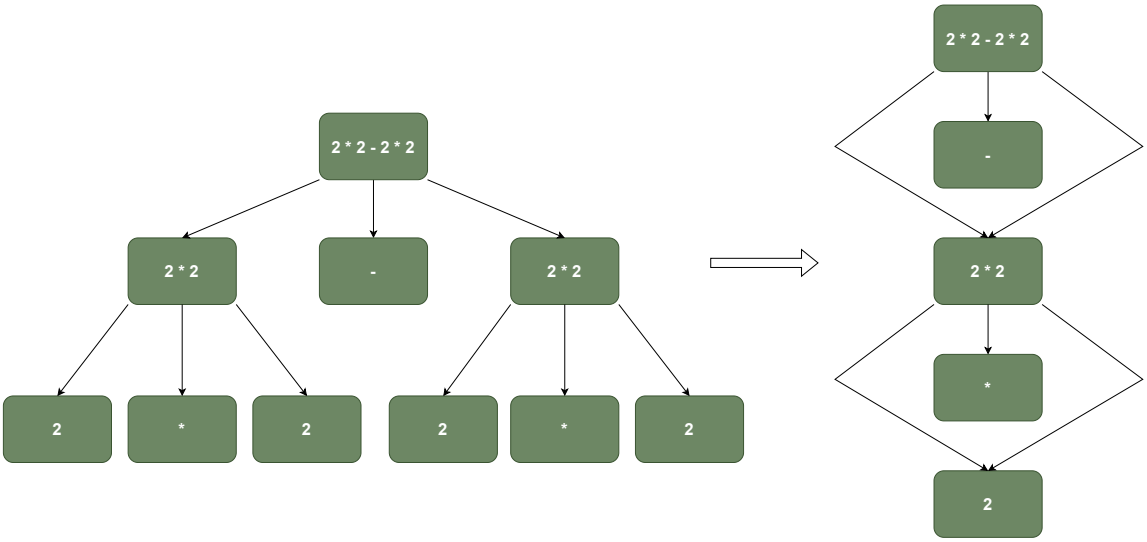
Vytvorenie parsera je inšpirované nástrojom `rust-analyzer` a reprezentáciou syntaktických stromov programovacieho jazyka Swift[5]. Jadrom je štruktúra *'Red-Green'* stromov, ktorá umožňuje efektívnym spôsobom ukladať uzly a navigáciu medzi nimi [20]. Architektúra *'Red-Green'* stromov pozostáva z dvoch vrstiev.

***'Green'* vrstva** - Predstavuje stromovú štruktúru. Táto vrstva obsahuje uložené dáta, kde každý uzol nesie pôvodnú textovú reprezentáciu zo zdrojového textu. Uzly sú netypované a je umožnené ich zdieľanie. Biele znaky a komentáre sú rovnako súčasťou tejto

[5] https://github.com/apple/swift/tree/5e2c815edfd758f9b1309ce07bfc01c4bc20ec23/lib/Syntax

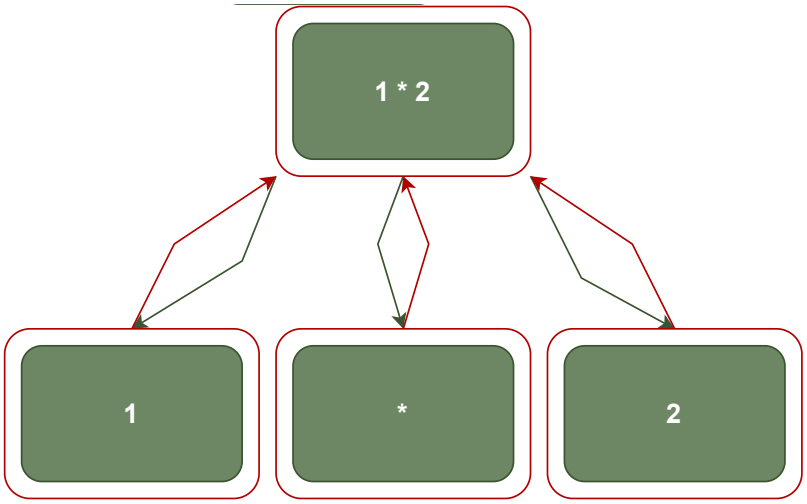
reprezentácie. Postupným spájaním jednotlivých tokenov je možné dostať plnohodnotnú reprezentáciu pôvodného textu. Chyba vo vstupnom texte je reprezentovaná chybovým uzlom. Každý uzol obsahuje väzbu na jeho priamych potomkov [20].

Uzly obsahujú okrem textovej reprezentácie aj dĺžku vstupného textu, ktorý pokrývajú. Dôležitou vlastnosťou uzlov je to, že je možné ich zdielať. Obrázok 5.3 demonštruje použitie zdieľania uzlov pre uloženie reprezentácie výrazu $2 * 2 - 2 * 2$. Lavá schéma reprezentuje výsledok parsovania a pravá schéma reprezentuje uloženie výslednej štruktúry v *'Green'* vrstve.



Obr. 5.3: Uloženia reprezentácie výrazu v syntaktickom strome za využitia zdieľania uzlov. Prevzaté z [26].

***'Red'* vrstva** - Túto vrstvu si je možné predstaviť ako akési obalenie *'Green'* vrstvy. Vzhľadom na to, že *'Green'* vrstva podporuje zdieľanie uzlov je potrebné, aby *'Red'* vrstva umožnila rozlíšiť jednotlivé uzly. Táto vrstva obsahuje navyše údaj o absolútnej pozícii daného uzla v texte, čím rozširuje pôvodnú reprezentáciu uzla, ktorá obsahovala len jeho dĺžku. Rovnako je pridaný ukazovateľ na predka daného uzla (respektíve tranzitívne na cestu od daného uzla až ku koreňovému uzlu). Tieto zmeny umožňujú jednoduché odkazovanie sa z výsledného syntaktického stromu na pôvodný text. Rovnako umožňujú aj jednoduchú navigáciu medzi uzlami. Tento koncept je neskôr v implementačnej časti rozšírený o tretiu vrstvu, ktorá poskytuje prístupové rozhranie nad týmto syntaktickým stromom. Obrázok 5.4 demonštruje zjednodušenú výslednú reprezentáciu *'Red-Green'* stromovej štruktúry.



Obr. 5.4: Obalenie *'Green'* vrstvy *'Red'* vrstvou.

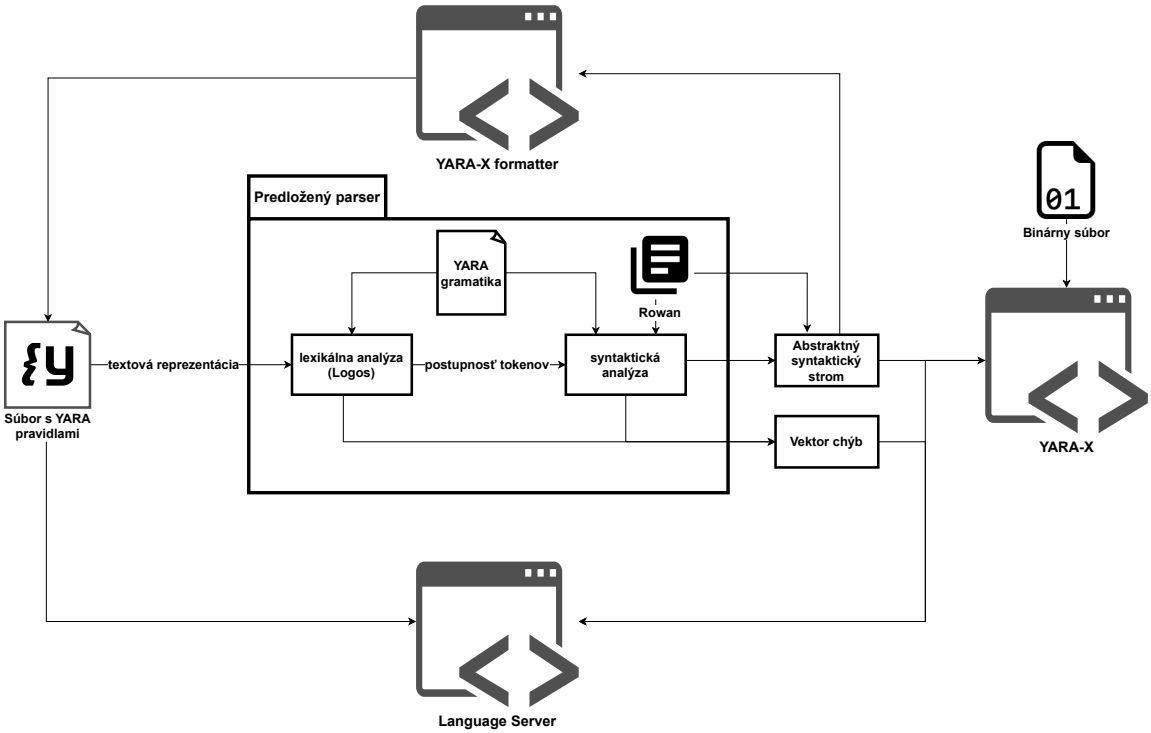
Popísaný koncept je implementovaný za využitia už existujúcej knižnice `rowan`[6], ktorá je mierne upravená pre účely tejto práce.

5.3.3 Architektúra parsera

Aktuálny parser nástroja YARA-X sa stará o lexikálnu a syntaktickú analýzu. Výstupom parsera je dvojica pravidlo a zodpovedajúca časť vstupného textu. Následne je manuálne zostrojený syntaktický strom, ktorý presne zodpovedá celému pôvodnému textu. Tento syntaktický strom je neskôr konvertovaný na abstraktný syntaktický strom, ktorý neobsahuje biele znaky, komentáre, aplikujú sa precedenčné pravidlá na operátory a zjednodušujú sa výrazy. Pri tvorbe abstraktného syntaktického stromu sa takisto aplikujú aj niektoré sémantické kontroly.

Navrhnutý parser nevykonáva žiadnu formu sémantickej kontroly. Pre potreby lexikálnej analýzy je využitá knižnica `logos`[7]. Výstup z tejto knižnice v podobe lexikálnych tokenov je predložený na vstup parsera. Parser tokeny postupne prejde, spracuje pomocou definovanej gramatiky a zostaví syntaktický strom. Jednotlivé uzly sú uložené v už spomenutej *'Red-Green'* stromovej štruktúre. Výstupom je výsledný syntaktický strom obsahujúci biele znaky, komentáre a aj zjednodušené výrazy pomocou precedencie operátorov. V prípade, že vstupný text obsahuje syntaktické chyby, výstupom môže byť aj nekompletný syntaktický strom s chybovými uzlami. Výstup dopĺňa aj vektor chýb, obsahujúci pozíciu a popis lexikálnych a syntaktických chýb. Výsledná štruktúra syntaktického stromu je využitá pri integrácii do projektu YARA-X a transformovaná do (H)IR medzipodoby. Kontrola sémantickej správnosti je plne presunutá a súčasťou tejto transformácie. Cieľom je poskytnúť takú reprezentáciu syntaktického stromu, ktorá okrem všetkej pridanej funkcionality zabezpečí aj jednoduchú podporu pôvodnej reprezentácie a integráciu do projektu YARA-X. Architektúra parsera odolného voči chybám je znázornená na obrázku 5.5. Konkrétnym implementačným detailom vytvoreného parsera a tvorby syntaktického stromu je venovaná kapitola 6.

[6] https://github.com/rust-analyzer/rowan
[7] https://docs.rs/logos/latest/logos/



Obr. 5.5: Architektúra predstaveného parsera a jeho využitie v ekosystéme YARA-X.

K demonštrácii funkcionality parsera a ním prinesených výhod je vytvorená jednoduchá implementácia jazykového servera, ktorá umožňuje napríklad zvýraznenie syntaktických chýb zdrojového textu, či navigáciu na definíciu premennej v danom pravidle.

Kapitola 6

Implementácia

Obsahom tejto kapitoly je popis implementácie viacerých rozšírení do projektu YARA-X. Práca sa zaoberá vytvorením a nasadením rozšírenia k zavedeniu podpory pre parsovanie prvého súborového formátu, konkrétne formátu Mach-O. Ďalej je predstavené nové rozšírenie, ktoré umožňuje využívať projekt YARA-X, aj ako nástroj pre extrakciu dát zo súboru a plynulo nadväzuje na vytvorený Mach-O modul. Záverečná sekcia implementačnej časti sa venuje implementácii nového parsera, ktorý je odolný voči chybám a adresuje potreby pre využitie nástroja YARA-X v integrovaných vývojových prostrediach. Súčasťou práce je aj vytvorenie jazykového servera pre demonštráciu funkcionality predstaveného parsera. Všetky navrhnuté riešenia významným spôsobom rozširujú aktuálnu architektúru a predstavujú dôležitý krok k využívaniu projektu YARA-X nie len ako jedného nástroja pre jednu konkrétnu činnosť, ale aj ako sadu nástrojov pomáhajúcu užívateľom a malvérovým analytikom pri ich každodenných potrebách.

Vytvorené rozšírenia a ich integrácia do už existujúceho projektu sú v čase písania práce komunikované nie len s firmou Gen Digital, ale aj so samotným autorom projektu YARA-X. Na základe jeho spätnej väzby je kód modifikovaný a vylepšovaný z dôvodu zaistenia jeho skorej integrácie do projektu. Pri popise jednotlivých častí je jasne uvedené, čo je vytvorené ako súčasť práce a na aké časti sa nadväzuje, prípadne čo je prevzaté a využité. Odovzdané zdrojové kódy obsahujú **README.md** súbor, ktorý detailnejšie popisuje ako projekt YARA-X zostaviť a upresňuje, ktoré časti boli vytvorené alebo modifikované ako súčasť tejto práce.

6.1 Mach-O modul

K implementovaniu rozšírenia pre podporu Mach-O formátu v nástroji YARA-X je využitá oficiálna dokumentácia[1] a kostra na vytváranie nových modulov. Vytvorenie nového modulu pozostáva z troch hlavných krokov:

1. Definícia protobuf štruktúry
2. Implementácia logiky modulu, ktorá sa stará o samotné parsovanie
3. Pridanie a exportovanie funkcií, ktoré sú dostupné z YARA pravidiel

[1] https://github.com/VirusTotal/yara-x/blob/main/docs/ModuleDeveloper'sGuide.md

6.1.1 Definícia protobuf štruktúry

Protobuf štruktúra predstavuje spôsob uloženia naparsovaných informácii o danom súbore pomocou konkrétneho typu modulu. Umožňuje jednoduchú serializáciu týchto informácii a ich sprístupnenie v YARA pravidlách. Podporované sú dve verzie, `proto2` a `proto3`.

Definícia štruktúry pre súbory typu Mach-O využíva predvolenú verziu `proto2`, keďže umožňuje niektoré polia definovať ako povinné, narozdiel od `proto3`, kde sú všetky polia voliteľné. Tento fakt umožňuje následne v YARA pravidlách využívať hodnotu `YR_UNDEFINED`. Na túto hodnotu sú prirodzene transformované všetky nenaplnené hodnoty polí v protobuf štruktúre, ktoré nie sú definované ako povinné a neobsahujú žiadnu hodnotu.

Súbor popisujúci vytvorenú štruktúru s názvom `macho.proto` je umiestnený v priečinku `/lib/src/modules/protos`. Tento priečinok obsahuje definície všetkých podporovaných modulov.

```
syntax = "proto2";

import "yara.proto";

option (yara.module_options) = {
    name: "macho"
    root_message: "macho.Macho"
    rust_module: "macho"
};
```

Hlavička protobuf súboru obsahuje okrem definície použitej verzie protobufu aj názov modulu, v tomto prípade `macho`. Nasleduje vloženie špecifických YARA definícii umiestnených v centrálnom protobuf súbore s názvom `yara.proto` a popis konkrétneho modulu. Popis Mach-O modulu obsahuje povinné polia `name` a `root_message` definujúce názov modulu a počiatočnú protobuf štruktúru. Názov je použitý pri vložení a použití modulu v YARA pravidlách pomocou kľúčového slova `import` nasledovaného názvom modulu `"macho"`. Počiatočná protobuf štruktúra definovaná ako `macho.Macho` označuje koreňovú správu, ku ktorej sa priamo pristupuje z YARA pravidiel. Nasledujúce správy, respektíve popisy vnorených štruktúr, sú z nej rekurzívne odkazované. Posledná časť `rust_module` popisuje názov príslušného modulu obsahujúceho kód, ktorý sa stará o logiku parsovania a plnenie tejto štruktúry. Táto funkcionalita je bližšie popísaná v sekcii 6.1.2 pričom existujú dva spôsoby, kde sa tento modul môže nachádzať:

- súbor `/lib/src/modules/macho.rs` pre jednoduché moduly
- priečinok `/lib/src/modules/macho/` obsahujúci `mod.rs` súbor a ďalšie dodatočné súbory pre kód alebo testy.

V práci je zvolený druhý spomínaný spôsob, vzhľadom na komplexnosť modulu a dodatočné jednotkové testy.

Nasledujú protobuf správy popisujúce očakávanú výslednú štruktúru pre súbory typu Mach-O. Koreňová správa `Mach-O` obsahuje polia hlavičky, ktoré popisujú základné informácie o súbore, ako napríklad hodnotu `magic`, jednoznačne identifikujúcu typ súboru alebo počet vnorených segmentov, či počet príkazov. Po hlavičke nasleduje viacero vnorených segmentov. Tieto segmenty popisujú časti binárneho súboru, ktoré sú načítané do pamäti.

Jedná sa o samotnú aplikáciu a pripojené knižnice. Segmenty môžu byť určené len na čítanie alebo aj na zápis. Každý takýto segment pozostáva z viacerých sekcií. Sekcie obsahujú priamo skompilovaný kód alebo iné časti programu.

V prípade, že sa jedná o FAT súbor, ktorý je určený pre viaceré architektúry hlavička obsahuje len jeho identifikáciu pomocou `magic` hodnoty a počet podporovaných architektúr. Nasledujú hlavičky pre každú z podporovaných architektúr a následne, rovnako ako pri bežnom Mach-O súbore, jednotlivé segmenty a sekcie.

Okrem definície štruktúry Mach-O súboru sú obsiahnuté aj rôzne konštanty, ktoré je možné využiť priamo z YARA pravidiel. Tieto konštanty sú definované ako zoznam hodnôt pre každú kategóriu. Napríklad zoznam s názvom `HEADER`

```
enum HEADER {
    option (yara.enum_options).inline = true;
    MH_MAGIC = 0 [(yara.enum_value).i64 = 0xfeedface];
    MH_CIGAM = 1 [(yara.enum_value).i64 = 0xcefaedfe];
    MH_MAGIC_64 = 2 [(yara.enum_value).i64 = 0xfeedfacf];
    MH_CIGAM_64 = 3 [(yara.enum_value).i64 = 0xcffaedfe];
}
```

obsahuje konštanty `MH_MAGIC, MH_CIGAM, MH_MAGIC_64 a MH_CIGAM_64`, ktoré definujú `magic` konštanty pre dané typy Mach-O súborov reprezentujúce 32 alebo 64 bitové architektúry a ich príslušnú endianitu. Za povšimnutie stojí, že hodnota nie je definovaná priamo, ale je použité protobuf rozšírenie `yara.enum_value`, ktoré je obsiahnuté práve v už spomínanej centrálnej YARA protobuf definícii `yara.proto`. Toto rozšírenie je použité z dôvodu, že protobuf neumožňuje v `enum` hodnotách využívať väčšiu ako 32 bitovú hodnotu, čo by nebolo postačujúce. Ďalšie z použitých rozšírení je `(yara.enum_options).inline`, ktoré umožňuje pristupovať k hodnotám typu `enum` priamo, bez nutnosti prefixu v podobe jeho názvu. Pre prístup k hodnote `MH_MAGIC` teda stačí v YARA pravidle použiť `macho.MH_MAGIC` miesto `macho.HEADER.MH_MAGIC`.

6.1.2 Implementácia logiky modulu

Logika parsovania Mach-O súboru a plnenie protobuf štruktúr prebieha v `mod.rs` súbore lokalizovanom v priečinku `/lib/src/modules/macho/`. Vstupným bodom je funkcia označená makrom `#[module_main]`. Táto funkcia je zavolaná pre každý súbor, ktorý je scanovaný nástrojom YARA-X pomocou pravidla, ktoré tento modul volá. Na vstupe do tejto funkcie je pole bytov s obsahom príslušného súboru. Výstupom je `Macho` štruktúra vygenerovaná protobuf súborom. Táto štruktúra sa postupne plní a poliam, ktoré sú v nej nadefinované je priradená získaná hodnota zo vstupného súboru.

V prvotnej fáze dochádza ku kontrole veľkosti vstupných dát. Pokiaľ ich veľkosť spĺňa minimálnu veľkosť Mach-O súboru, a teda obsahuje aspoň hlavičku, ktorá ide naparsovať, dochádza ku kontrole či sa jedná o FAT súbor. Táto kontrola prebieha pomocou získania a overenia `magic` hodnoty. Parsovanie FAT súborov prebieha podobne ako parsovanie súborov určených pre jednu architektúru. Jediným rozdielom je počiatočné získanie hlavičiek pre každú architektúru a zavolanie parsovania Mach-O súbora pre každý vnorený súbor. Ich počiatočný index vo vstupnom súbore je získavaný z FAT hlavičky a konečný index pomocou pričítania veľkosti daného súboru, ktorá je taktiež obsiahnutá vo FAT hlavičke.

Následné pokračovanie parsovania už prebieha rovnakým spôsobom ako pre bežné súbory. Funkcia `parse_macho_file()` sa stará o získanie dát z hlavičky súbora, prevod bytov

do *little-endian* formátu ak sa v ňom ešte nenachádzajú, naplnenia protobuf polí definujúcich hlavičku súboru konkrétnymi hodnotami a zavolania funkcie pre parsovanie vnorených segmentov.

K parsovaniu bytov a získaniu príslušných hodnôt či už pre hlavičku, segmenty, alebo vnorené sekcie je využitá knižnica `nom`. Táto knižnica umožňuje postupné spracovanie vstupných dát a extrakciu požadovaných hodnôt. Pomocou metód tejto knižnice je možné iteratívne parsovať vstupné dáta a získať požadované údaje, ktoré sú následne využité k naplneniu protobuf štruktúry. Táto štruktúra poskytuje metódy `set_<foo>` a `get_<bar>` k nastaveniu, respektíve získaniu aktuálnej hodnoty pre dané pole. V prípade nastavenia hodnoty `magic`, ktorá odpovedá polu `Macho.magic` je použitá metóda `macho_file.set_magic()`. Premenná `macho_file` predstavuje inicializovaný protobuf súbor s koreňovou štruktúrou `Macho`. Funkcia `set_magic()` berie ako argument `parsed_header.magic`, čo je premenná predstavujúca výsledok naparsovanej hlavičky vstupného súbora pomocou knižnice `nom`.

Spracovanie segmentov je rozdelené na dve časti. V prvej časti sa skontroluje počet aktuálne podporovaných segmentov. K získaniu tohto počtu sa využije počítadlo, ktoré sa inkrementuje po preskočení špecifického počtu bytov predstavujúceho veľkosť segmentu. Nasleduje získanie údajov o danom segmente. Tento proces je podobný ako pri získavaní informácii o hlavičke súbora. Každý segment sa skladá z viacerých sekcií. Sekcie sú postupne spracované a naplnia sa hodnoty daných segmentov spolu s vnorenými sekciami v protobuf súbore. Ostatné spracované príkazy, ktoré nie sú považované za segmenty sú `LC_UNIXTHREAD` a `LC_MAIN`. Prvý menovaný príkaz popisuje vstupný bod binárneho súboru pre staršie architektúry. Tento príkaz bol neskôr nahradený príkazom `LC_MAIN`. Počas spracovania oboch uvedených príkazov sa aktuálna pozícia v súbore konvertuje na relatívnu virtuálnu adresu (RVA). Táto adresa je nezávislá na špecifickom umiestnení programu v pamäti.

6.1.3 Exportovanie funkcií

Poslednou súčasťou Mach-O modulu je možnosť exportovania funkcií. Okrem umožnenia prístupu k definovaným štruktúram daného modulu existuje aj možnosť volať funkcie priamo z YARA pravidiel. Táto funkcionalita je obsiahnutá vo viacerých moduloch v pôvodnom nástroji YARA.

V nástroji YARA-X existuje atribút `#[module_export]`. Tento makro výraz umožňuje definovať zvolenú funkciu ako exportovanú funkciu. Definícia exportovanej funkcie obsahuje aspoň jeden povinný argument, obsahujúci všetky potrebné informácie o kontexte spracovávaného súboru. Zvyšné argumenty sú voliteľné a viditeľné aj priamo pri volaní danej funkcie z YARA pravidiel. Rozšírením atribútu `module_export` je možnosť definovať vlastné meno funkcie pomocou parametru `name=<vlastne_meno_funkcie>`. Táto možnosť je užitočná pri preťažovaní funkcií. Jedna z exportovaných funkcií v Mach-O module má názov `file_index_for_arch`.

```
#[module_export(name = "file_index_for_arch")]
fn file_index_type(
    ctx: &mut ScanContext,
    type_arg:  i64,
) ->  Option< i64> {...}
```

```
#[module_export(name = "file_index_for_arch")]
fn file_index_type(
    ctx: &mut ScanContext,
    type_arg:  i64,
     subtype_arg: i64,
) ->  Option< i64> {...}
```

Táto funkcia obsahuje dve definície s rôznymi názvami a počtom parametrov. Pri jej využití v YARA pravidlách sa však jedná o rovnakú funkciu. Príslušná definícia sa zavolá v závislosti na tom, koľko parametrov je dodaných pri zavolaní funkcie s názvom `file_index_for_arch()` z YARA pravidla. Funkcia slúži na získanie indexu vnoreného Mach-O súboru pre binárne súbory typu FAT v závislosti na špecifikovanom type alebo podtype procesoru. Ďalšou z exportovaných funkcií, ktoré sú podporované v Mach-O module je `entry_point_for_arch()`. Jedná sa o funkciu, ktorá umožňuje získanie adresy vstupného bodu súboru pre špecifický typ alebo podtyp procesoru.

6.2 YARA-X ako nástroj pre extrakciu dát

V poradí druhým implementovaným rozšírením je nástroj pre extrakciu dát. Cieľom tohto nástroja je umožniť jednoduchšiu vizualizáciu a prácu s informáciami o súbore, ktoré boli nadobudnuté pri použití jedného z modulov na parsovanie súborov. Toto rozšírenie priamo nadväzuje na implementovaný Mach-O modul. Vytvorené je univerzálne rozšírenie pre nástroj YARA-X, ktoré funguje ako samostatný celok. Hlavná motivácia pre vytvorenie rozšírenia pre extrakciu dát zo súboru je zamedzenie využívania externých nástrojov pri analýze súborov.

6.2.1 Architektúra nástroja pre extrakciu dát

Implementáciu rozšírenia je možné rozdeliť na dva logické celky. Prvý celok sa stará o pripravenie prostredia v projekte YARA-X a extrahovanie informácii z protobuf štruktúry. Úloha druhého celku je tieto informácie spracovať, transformovať do požadovaného formátu a poskytnúť užívateľovi v čo najprívetivejšej podobe. Pôvodne YARA obsahovala možnosť ako získať informácie zo súborov. Tento výstup, ktorý je znázornený na obrázku 6.1 bol však pomerne neprehľadný a obmedzený.

```
macho
        file
        fat_arch
        nfat_arch = YR_UNDEFINED
        fat_magic = YR_UNDEFINED
        stack_size = YR_UNDEFINED
        entry_point = 116
        segments
                [0]
                        segname = "SP1\xc0\x89\xe7j\x08Wj\x01P\xb0\x04\xeb\xb0"
                        vmaddr = 4096
                        vmsize = 4096
                        fileoff = 0
                        fsize = 164
                        maxprot = 7
                        initprot = 5
                        nsects = 0
                        flags = 0
                        sections
        number_of_segments = 1
        reserved = YR_UNDEFINED
        flags = 18874368
        sizeofcmds = 136
        ncmds = 2
        filetype = 2
        cpusubtype = 3
        cputype = 7
        magic = 4277009102
```

Obr. 6.1: Pôvodný výstup modulov programu YARA pre výpis informácii zo súboru.

Implementácia prvej časti je riešená v spolupráci s Victorom M. Alvarezom. Pri prvotnej implementácii tejto časti sa zistila skutočnosť, že YARA nedisponuje možnosťou spustiť moduly na parsovanie súboru bez toho, aby boli súčasťou pravidla. V praxi to znamená nutnosť vždy vytvoriť akýsi *'dummy'* súbor. Tento súbor nehrá v samotnom procese žiadnu rolu a jedná sa o YARA pravidlo, ktorého jedinou úlohou je vloženie a spustenie príslušného modulu pomocou kľúčového slova `import`. Prvá verzia tohto rozšírenia teda pracuje s týmto faktom a vždy vytvorí 'dummy' pravidlo. Jedným zo stanovených cieľov pri vytvorení modulu pre extrakciu dát je okrem jeho univerzálnosti aj robustnosť. Prvá vytvorená verzia sa nejavila úplne ideálne a s touto skutočnosťou bol konfrontovaný aj Victor Alvarez. Spoločne sme dospeli k záveru, že bude potrebné prispôsobiť aj nástroj YARA-X na tento modul.

Jeho úloha pri tomto rozšírení spočíva v pripravení nového prístupového bodu, ktorý umožní získať informácie vyprodukované ľubovolným modulom [2]. Moja práca spočíva v upravení modulu a jeho napojení na novú architektúru.

6.2.2 Napojenie CLI prostredia

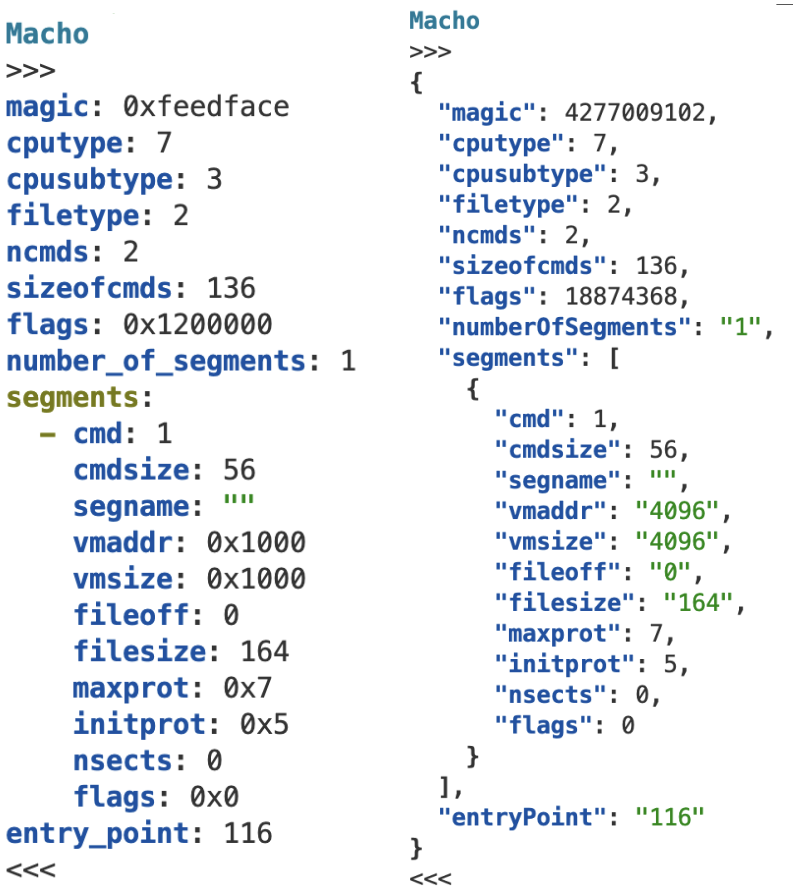
Vzhľadom na fakt, že rozšírenie je navrhnuté ako samostatný celok, ktorý na vstupe zoberie protobuf štruktúru a transformuje ju do serializovateľného a užívateľsky prívetivého formátu je potrebná aj jeho integrácia do projektu YARA-X. Integrácia spočíva v pridaní nového argumentu pre **C**ommand **L**ine **I**nterface prostredie nástroja YARA-X.

[2] https://github.com/VirusTotal/yara-x/pull/52

YARA-X disponuje samostatným kontajnerom pre CLI prostredie s názvom `cli`. Táto časť spracováva vstupné argumenty a volá im príslušné funkcie, respektíve moduly. Keďže úlohou implementovaného rozšírenia je extrakcia a sprostredkovanie dát užívateľovi, vstupný argument pre túto funkcionalitu dostal názov `dump`. Nasleduje voliteľný názov súboru. Pokiaľ názov alebo cesta k súboru nie je prítomná, spracováva sa štandardný vstup vložení na `STDIN`. Pokračujú voliteľné argumenty `-o` alebo `-output-format` pre špecifikáciu výstupného formátu a `-c` respektíve `-color` pre vytvorenie farebne odlíšiteľnej verzie výstupu. Posledným nepovinným argumentom je `-m` alebo `-modules`, ktorý poskytuje možnosť špecifikácie požadovaného modulu, ktorý sa na daný súbor zavolá.

Práve v tejto časti nastáva napojenie na prístupový bod, ktorý bol implementovaný Victorom M. Alvarezom. Tento bod pozostáva z funkcie **invoke_mod_dyn::<module>()**, ktorá zavolá príslušný modul a vráti jeho výstup. Súčasťou implementácie je aj definícia podporovaných modulov. Nástroj YARA-X obsahuje viacero modulov, ale nie pri všetkých z nich dáva zmysel spracovávať výstup. Moduly ako `time` alebo `console` slúžia len k doplneniu funkcionality do jazyka YARA a žiadnym spôsobom nepracujú so vstupnými súbormi a ani neprodukujú žiadny výstup v podobe protobuf štruktúr. V súčasnej dobe podporované moduly pre spracovanie súborov sú **Mach-O, ELF, LNK** a **PE**. Volanie príslušného modulu môže byť manuálne, pomocou špecifikácie modulu argumentom `modules` alebo automatické. Manuálna špecifikácia modulu umožňuje aj zavolanie viacerých modulov v prípade neznámeho alebo poškodeného súboru. Implementované rozšírenie vyhodnotí napríklad pomocou *magic* hodnôt, ktoré z výstupných štruktúr dávajú zmysel a poskytne ich užívateľovi. V prípade chýbajúceho argumentu `modules` rozšírenie funguje v móde automatického výberu. Tento mód spustí všetky podporované moduly, vyfiltruje výstupy, ktoré nedávajú zmysel a zvyšné zobrazí. Pre každý modul je zavolaná funkcia `invoke_mod_dyn::<module>()`, ktorá vráti naplnenú protobuf štruktúru pre daný modul. Filtrovanie a automatický výber spočíva v hľadaní príznaku v protobuf štruktúrach, ktorý musí byť nastavený, prípadne v hľadaní pola, ktorého hodnota musí byť nastavená. Tento príznak môže byť napríklad `has_magic()` v prípade Mach-O formátu súboru alebo `is_pe()` pri formáte PE.

Poslednou časťou rozšírenia v CLI sekcii je funkcia `obtain_module_info()`, ktorá slúži k zavolaniu nástroja na transformáciu protobuf štruktúry do jedného z podporovaných formátov a poskytnutie výstupu užívateľovi. Aktuálne podporované formáty výstupu sú dva: **JSON** a **YAML**. Pričom druhý menovaný je aj predvolený formát. JSON aj YAML predstavujú moderné formáty, ktoré patria medzi najviac využívané formáty určené k serializácii dát v súčasnosti. Poskytujú výstup, ktorý môže byť jednoducho strojovo spracovateľný, prenášaný, uložený, či využívaný inými externými nástrojmi. V prípade potreby len jednoduchého zobrazenia výstupu je pridaná aj možnosť využitia farebného rozlíšenia jednotlivých sekcii pomocou argumentu `color`. Obrázok 6.2 znázorňuje príklad výsledného zobrazenia informácii o súbore v užívateľsky prívetivom prevedení pre obidva formáty. Ľavá časť obrázku predstavuje YAML formát a pravá formát JSON.



Obr. 6.2: Výsledné zobrazenie informácii o súbore v užívateľsky prívetivej podobe pomocou nástroja pre extrakciu dát zo súboru.

6.2.3 Transformácia protobuf štruktúry na strojovo spracovateľný formát

Druhá logická časť rozšírenia nástroja pre extrakciu dát spočíva v transformácii protobuf štruktúry na jeden zo zvolených strojovo spracovateľných formátov. Táto časť funguje ako samostatný celok nezávisle na YARA-X architektúre. Po dohode s autorom nástroja YARA-X je umiestnená priamo do YARA-X prostredia ako samostatný kontajner s názvom `proto-yaml`. Súčasťou kontajneru sú okrem zdrojových súborov aj jednotkové testy a `.proto` definícia.

Jedným z ďalších cieľov návrhu je implementovanie funkcionality, ktorá umožní zobraziť výstup aj v užívateľsky prívetivej podobe. Využíva sa rozšírenie YAML formátu, ktorý ponúka väčšie možnosti úpravy a prispôsobenia výstupu. Vizuálne prívetivý formát nadväzuje na možnosť farebného rozšírenia jednotlivých častí a pridáva možnosť doplnenia komentárov, ktoré sú priamo podporované YAML formátom a môžu slúžiť k doplneniu významu jednotlivých sekcií. Ďalším rozšírením je schopnosť reprezentácie špecifikovaných číselných hodnôt nie len ako decimálnych, ale aj ako hexadecimálnych, prípadne ako hodnôt časových razítok. Možnosť označenia toho, ktoré hodnoty sa majú zobraziť v inom ako predvolenom formáte, je špecifická pre každý modul a nachádza sa priamo v jeho `.proto` definícii.

```
message Macho {
     // Set Mach-O header and basic fields
    optional uint32 magic = 1 [(yaml.field).fmt = "x"];
    optional uint32 cputype = 2;
    ...
}
...
message Dylib {
    optional string name = 1;
    optional uint32 timestamp = 2 [(yaml.field).fmt = "t"];
    optional string compatibility_version = 3;
    optional string current_version = 4;
}
...
```

Parameter `fmt` je pridaný ako súčasť rozšírenia popisu nastavení jednotlivých polí. Táto možnosť aktuálne umožňuje dve špecifikácie: **'x'** a **'t'**. V príklade uvedenom vyššie je použitý parameter **'x'** k špecifikácii, že hodnota pola `magic` vo výstupnom formáte má byť zobrazená ako hexadecimálna a nie decimálna. V prípade hodnoty pola `timestamp` je použitý parameter **'t'**, ktorý vo výstupe pri danej hodnote okrem jej decimálnej reprezentácie doplní aj komentár obsahujúci čitateľnú podobu časového razítka. Využitie uvedených formátov pre decimálne hodnoty je demonštrované na obrázku 6.3. Jedná sa o výstup rozšírenia pre extrakciu dát zo súborov. Vstupný súbor bol manuálne vytvorený. Pre hodnotu pola `int32_hex` bol použitý parameter **'x'** a pre pole `timestamp` bol použitý parameter **'t'**.

```
int32_hex: 0x7b
timestamp: 1703964702 # 2023-12-30 19:31:42 UTC
int32_dec: 123
str: "foo"
repeated_msg:
  - int32_dec: 456
    str: "bar\nbar"
  - int32_dec: 789
    str: "baz"
    map_string_string:
        "foo\nbar": "bar"
nested_msg:
    int32_dec: 1234
    str: "qux\nfoo"
    map_string_string:
        "bar": "baz"
```

Obr. 6.3: Využitie rôzneho formátu pre decimálne hodnoty.

Prevod protobuf štruktúry do YAML formátu je priamočiary a využije sa knižnica `protobuf_json_mapping`. Zavolá sa funkcia `print_to_string()` a prevedie protobuf reprezentáciu do JSON podoby. Prevod do YAML formátu je už o niečo zložitejší a prebieha priamo v kontajneri `proto-yaml`.

O prevod do formátu YAML sa stará štruktúra `Serializer` obsahujúca prístupový bod v podobe funkcie `serialize`. Na vstupe tejto funkcie je protobuf štruktúra a výstupom je korešpondujúca YAML reprezentácia. Princíp spočíva v postupnom prechode protobuf štruktúry, ktorá pozostáva z troch základných typov polí: voliteľné pole, povinné pole a slovník (dvojica kľúč-hodnota). Prvky Protobuf štruktúry sa postupne prechádzajú hodnota za hodnotou a každý typ je spracovaný samostatne. Udržuje sa informácia o aktuálnom odsadení, teda počte bielych znakov oddeľujúcich začiatok riadku vo výstupe od danej hodnoty. Názov pola je spracovaný a uložený pomocou funkcie `write_field_name()`, ktorému predchádza alebo je nasledovaný príslušným odsadením. Spracovanie pokračuje zápisom hodnoty pola pomocou funkcie `write_value()`, za ktorou takisto môže nasledovať odsadenie. Hodnoty môžu nadobúdať rôzne či už číselné, textové alebo zložené hodnoty. V prípade opakovaných hodnôt je vložený nový riadok medzi názvom pola a jemu odpovedajúcim hodnotám.

Dôležitou súčasťou implementácie je funkcia `print_integer_value_with_options()`, ktorá využíva nastavenia polí v protobuf štruktúre a transformuje číselnú reprezentáciu do hexadecimálnej podoby alebo takisto do čitateľnej podoby časového razítka. Štruktúra `ColorsConfig` obsahuje nastavenia farieb pre jednotlivé časti výstupného formátu. Pokiaľ je nastavený príznak farebného výstupu, spracovanie jednotlivých častí využíva túto štruktúru k získaniu farebnej reprezentácie a jej aplikovaniu na danú hodnotu.

6.3 Parser odolný voči chybám

Súčasťou rozšírenia ekosystému nástroja YARA-X je aj vybudovanie parsera, ktorý bude odolný voči chybám. Cieľom je nahradiť aktuálny parser a poskytnúť univerzálne riešenie, ktoré bude možné využiť aj externými nástrojmi. Výsledné riešenie umožní vytvoriť abstraktný syntaktický strom aj pre chybný alebo nekompletný vstup. Aktuály parser toto neumožňuje. Jednou z hlavných nevýhod aktuálneho riešenia je jeho neschopnosť spracovať celý vstup. Pri narazení na prvú chybu je parsovanie ukončené ako neúspešné a vráti sa chybová hláška.

Dôležitým aspektom pri budovaní parsera odolného voči chybám je aj jeho integrácia do už existujúceho prostredia. Z tohto dôvodu bolo nutné v určitých ohľadoch dospieť ku kompromisu pre potreby práce a zároveň jednoduchej integrácie a napojenia na ostatné komponenty ekosystému. Vytvorený parser funguje ako samostatný kontajner s vlastnými závislosťami a je plne nezávislý na YARA-X projekte. Parser spracuje vstup, ktorý je poskytnutý ako textový reťazec. Výstupom je štruktúra `Parse<T>`, ktorá pomocou metódy `tree()` poskytuje abstraktný syntaktický strom reprezentujúci vstupný text a množinu chýb odhalených pri parsovaní vstupu. Táto množina je prístupná pomocou metódy `errors()`. Výsledná štruktúra dovoľuje jednoduchú integráciu s aktuálnym riešením.

V prípade syntakticky správneho formátu vstupu je možné pracovať s poskytnutým abstraktným syntaktickým stromom obdobným spôsobom ako pri jeho pôvodnej verzii, ktorá bola poskytnutá predchádzajúcim parserom. Tento abstraktný syntaktický strom je postupne spracovaný od koreňa k listom a jednotlivé uzly, reprezentujúce prvky vstupného textu sú prevedené do medzipodoby (IR), ktorá narozdiel od abstraktného syntaktického stromu obsahuje aj typové informácie. Táto podoba je vzdialenejšia od vstupného textu a využíva sa k vytvoreniu skompilovaných pravidiel obsiahnutých vo vstupnom texte. Pôvodný parser nemal oddelenú sémantickú kontrolu od syntaktickej. Kontrola sémantiky vstupného textu prebiehala či už na úrovni parsera, alebo práve pri prevode abstraktného syntaktického stromu do medzipodoby určenej ku kompilácii.

V prípade chybného vstupu je možné zobraziť celú množinu chýb na vstupe. Táto množina obsahuje okrem prvotnej chyby aj zvyšné chyby, ktoré sa vyskytli pri syntaktickej kontrole zvyšnej časti vstupu. Rozširuje to tak schopnosti predchádzajúceho parsera, pričom funkcionalita ostáva zachovaná. Ostatné nástroje, ktoré či už sú, alebo nie sú súčasťou ekosystému budú schopné využiť poskytnutý abstraktný syntaktický strom obsahujúci aj chybové uzly.

6.3.1 Požiadavky na parser

Pri implementácii nového parsera je kladený dôraz na dva hlavné aspekty:

1. Jednoduchá integrácia s aktuálnym riešením a zachovanie úplnej aktuálnej funkcionality.
2. Vytvorenie vylepšeného riešenia, ktoré uspokojí požiadavky iných nástrojov, ktoré pre spracovanie jazyka YARA využívali vlastný parser.

Každý z bodov prináša so sebou istú mieru dodatočných požiadaviek, na ktoré je nutné vo výslednom riešení brať ohľad. Pre splnenie prvého bodu bola potrebná intenzívna komunikácia s Victorom M. Alvarezom. Jeho hlavným cieľom bolo vypracovanie podobnej štruktúry abstraktného syntaktického stromu tak, aby neboli nutné veľké zmeny v zvyšných komponentoch architektúry YARA-X. Tento bod so sebou prináša poskytnutie metód nad jednotlivými úrovňami stromu, ktoré jednoduchým spôsobom sprístupnia textovú reprezentáciu pre daný uzol v texte. Rovnako sú dostupné aj funkcie na hierarchické prechádzanie abstraktného syntaktického stromu, pomocou ktorých je možné sprístupniť či už nasledovníkov, alebo predchodcov daného uzlu. Ďalšou z požiadaviek bolo jednoduché napojenie na chybový systém v projekte YARA-X a možnosť reprezentovať vzniknuté chyby jednotným spôsobom.

Druhý bod cieli na možnosti práce s výslednou reprezentáciou vstupného textu. Nástroje ako YARI alebo YLS využívajú vlastný parser, ktorý odďaľuje sémantické kontroly a snaží sa o imitáciu parsovania odolného voči chybám. Vybudovanie jednotného parsera poskytne možnosť tieto nástroje nahradiť alebo integrovať do ekosystému. Ďalšími nástrojmi, ktoré v budúcnosti môžu rozšíriť ekosystém, sú napríklad nástroj na automatickú opravu chybných YARA pravidiel, nástroj na tvorbu, parsovanie či používanie YARA pravidiel v iných programovacích jazykoch.

Vzhľadom na nastavené požiadavky výsledná implementácia parsera odolného voči chybám kladie dôraz na:

1. Jednoduchú integráciu so zvyškom komponentov nástroja YARA-X;
2. Poskytnutie typovaného systému chybových hlášok, ktorý bude obsahovať všetky chyby vzniknuté na vstupnom texte;
3. Vytvorenie abstraktného syntaktického stromu, ktorý spĺňa nasledujúce požiadavky:
    - Korešpondencia so vstupným textom (biele znaky aj komentáre budú jeho súčasťou);
    - Jednoduchá navigácia na potomkov a predkov daného uzlu;
    - Jednoduchý prístup k textovej reprezentácii daného uzlu;
    - Možnosť jednoduchej aktualizácie stromu;

- Reprezentácia chybových stavov aj s miestom výskytu.

Pre overenie správnosti požiadaviek a integrácie vzniknutého parsera bol vytvorený jeho prototyp pre podmnožinu jazyka YARA.

6.3.2 Parser podmnožiny jazyka YARA

Prvým stavebným kameňom celej architektúry vzniknutého parsera je vytvorenie jeho prototypu, ktorý spracováva len pomerne malú podmnožinu jazyka YARA. Tento prototyp umožňuje spracovanie pravidiel, ktoré obsahujú jednoduché textové reťazce v sekcii `strings` a v sekcii `condition` sa môžu vyskytovať len pravdivostné výrazy. Tieto výrazy podporujú operátori `AND` a `OR`. Na mieste operandov sa môžu vyskytovať len premenné deklarované v sekcii `strings` alebo pravidvostné hodnoty `true` a `false`. Napríklad podporované pravidlo môže vyzerať nasledovne:

```
rule foo {
  strings:
    $a = "foo"
    $b = "bar"
 condition:
    $a and $b or true
}
```

Reprezentácia pravidiel podporovaných danou gramatikou pre podmnožinu jazyka YARA je definovaná následovne:

```
SOURCE -> RULE | eps.
RULE -> rule identifier lbrace RULEBODY rbrace.
RULEBODY -> STRINGS CONDITION | CONDITION .
STRINGS -> string colon STRINGSBODY.
CONDITION -> condition colon EXPRESSION.
STRINGSBODY -> variable assign string STRINGSBODY | eps.
EXPRESSION -> LITERAL EXPRESSION_TAIL.
EXPRESSION_TAIL -> OPERATOR EXPRESSION EXPRESSION_TAIL | eps.
LITERAL -> variable | BOOLEAN.
BOOLEAN -> true | false.
OPERATOR -> and | or.
```

Táto gramatika je typu LL(1). To znamená, že gramatika generuje vety prečítaním symbolov zľava doprava a vždy vyberá najľavejší symbol na expanziu. Číslo `1` znamená, že pri výbere nasledujúcej expanzie (derivačného kroku) sa stačí pozrieť na jeden symbol dopredu. Tento typ gramatiky sa využíva pri prístupe rekurzívneho zostupu pri syntaktickej analýze. Tento prístup bol zvolený na základe jeho vhodnosti pri tvorení parsera odolného voči chybám a jednoduchosti na implementáciu. Pri narazení na chybu vo vstupnom texte je daná chyba zaznamenaná spolu s jej výskytom a nasleduje zotavenie z chyby. Pre zotavenie z chyby práca volí využitie množiny `First`. Množinu First(x)[3] definujeme ako množinu všetkých terminálov, ktorými môže začínať reťazec derivovateľný z x. Vo vyššie uvedenom

[3] https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj07-en.pdf

príklade sú neterminálne symboly popísané slovom obsahujúcim len veľké písmená a terminálne symboly slovom obsahujúcim len malé písmená. V prípade zotavenia z chýb to znamená nastavenie synchronizačného bodu pri vzniku chyby na jeden z validných výrazov danej časti. V prípade výskytu chyby pri parsovaní v časti `STRINGS` je vypočítaná využitá množina `First` obsahujúca terminál `string` ako synchronizačný bod. To v praxi znamená, že sa generuje chyba a preskakuje všetko, až kým sa nenarazí na jeden z definovaných terminálov. Ukázalo sa, že v niektorých prípadoch je nutné rozšírenie tejto množiny aj o množinu `Follow`. Množina Follow(A)[4] je definovaná ako množina všetkých terminálov, ktoré sa môžu vyskytovať vpravo od A vo vetnej forme. Jedná sa teda o rozšírenie množiny o terminál `condition`. Výsledkom je množina `string a condition`, ktorá predstavuje zjednotenie množín `First` a `Follow`. Prvky tejto množiny predstavujú očakávané synchronizačné body pre zotavenie.

Architektúra prototypu parsera pre podmnožinu jazyka YARA je totožná s architektúrou výsledného riešenia predstaveného na obrázku 5.5, keďže tento prototype je postupne rozširovaný o prvky celého jazyka.

6.3.3 Lexikálna analýza

Aktuálne projekt YARA-X využíva knižnicu Pest, ktorá sa stará o lexikálnu aj syntaktickú analýzu. Súčasťou tvorby abstraktného syntaktického stromu sú aj niektoré sémantické kontroly. Vo výslednej podobe AST tým pádom nie je úplne oddelená syntax od sémantiky.

Predstavená implementácia parsera odolného voči chybám disponuje aj vlastnou lexikálnou analýzou, ktorá je plne oddelená a nezávislá od syntaktickej kontroly a tvorby abstraktného syntaktického stromu. Implementácia lexikálnej analýzy je priamo súčasťou predstaveného parsera. Tento parser je samostatný kontajner obsahujúci zdrojové kódy napísané v jazyku Rust. Časť starajúca sa o lexikálnu analýzu je umiestnená v priečinku s názvom `lexer`. Súčasťou priečinku je len jediný zdrojový súbor s názvom `mod.rs`, ktorý okrem zdrojového kódu určeného pre lexikálnu analýzu obsahuje aj jednotkové testy.

Implementácia lexikálnej analýzy je riešená pomocou knižnice `logos`[5]. Dôležitým aspektom pri výbere danej knižnice je rýchlosť a jednoduchosť používania. Tvorba tokenov zo vstupného textu je riešená pomocou štruktúry `LogosToken`. Táto štruktúra obsahuje definíciu všetkých podporovaný tokenov, ktoré sa vo vstupnom texte môžu nachádzať.

```
pub(crate) enum LogosToken {
    Import,
    #[token("rule")]
    Rule,
    // Identifiers
    #[regex(r"[a-zA-Z0-9_]*", |lex| lex.slice().to_string())]
    Identifier(String),
    // Variables
    #[regex(r"$[a-zA-Z0-9_]*", |lex| lex.slice().to_string())]
    Variable(String),
    ...
}
```

[4] https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj07-en.pdf
[5] https://logos.maciej.codes

Vyššie definovaný príklad štruktúry tokenov podporuje 2 možné typy ich reprezentácie. Prvý typ je možné zadefinovať priamo pomocou kľúčového slova `token`. V tomto prípade sa vo vstupnom texte hľadá definovaný reťazec. Napríklad pre prípad `#[token("rule")]` sa hľadá reťazec 'rule' a následne sa vytvorí token `Rule`. Druhý možný prípad je definícia hľadaného reťazca pomocou regulárneho výrazu. V prípade tvorby tokenu pre identifikátory jazyka YARA sa používa regulárny výraz `r"[a-zA-Z0-9_]*"`. Tento výraz znamená, že sa vo vstupnom texte hľadajú reťazce, ktoré obsahujú ľubovoľný počet opakovaní číslic , malých alebo veľkých písmen abecedy či podčiarkovník. V prípade lexikálnej analýzy nepodporovanej časti vstupu, teda časti, pre ktorú nie je pomocou žiadneho z definovaných pravidiel možné vytvoriť odpovedajúci token, je vytvorená reprezentácia chybového stavu. Tento stav obsahuje chybovú hlášku a miesto výskytu vo vstupnom texte. Neskôr v kóde je tento chybový stav transformovaný na verziu syntaktickej chyby s odpovedajúcou štruktúrou.

Jadro lexikálnej analýzy je implementované pomocou funkcie `tokenize()`. Táto funkcia berie na vstup reťazec reprezentujúci vstupný text a výstupom je množina tokenov a množina prípadných syntaktických chýb, ktoré vznikli pri tvorbe tokenov. Ako bolo už spomenuté, lexikálne chyby sú riešené ako istý podtyp syntaktických chýb. Funkcia `tokeniza` získa množinu tokenov pomocou knižnice `Logos` a ich definície v štruktúre `LogosToken`. Táto forma reprezentácie je však pre parser nepostačujúca a musí dôjsť ku konverzii z typu `LogosToken` na štruktúru `Token`. Táto štruktúra obsahuje dva atribúty: `kind` a `len`. Prvý z menovaných reprezentuje typ tokenu a jedná sa o priamu konverziu z typu `LogosToken` na `SyntaxKind`. `SyntaxKind` je typ tokenu, s ktorým implementovaný parser dokáže pracovať. Jedná sa o množinu terminálnych alebo neterminálnych symbolov, ktoré sa vyskytujú v gramatike pre daný jazyk. V predloženej práci ide o gramatiku pre jazyk YARA. Tvorbe množiny `SyntaxKind` a gramatike, ktorá ju definuje, sa bude práca venovať bližšie v nasledujúcej kapitole. Druhým atribútom je `len`, ktorý reprezentuje len dĺžku daného tokenu. K spomínanej konverzii dochádza počas cyklu prechádzajúceho všetky získané tokeny. Z každého získaného tokenu `LogosToken` je extrahovaná jeho dĺžka a spolu s jeho typom je po typovej konverzii uložená do štruktúry `Token`. V prípade výskytu chyby je vytvorená chybová hláška spolu s tokenom `SyntaxKind::ERROR`, ktorý chybu reprezentuje. Následne sú všetky tokeny zozbierané a spolu s chybami vrátené ako výstup funkcie.

Zaujímavým aspektom lexikálnej analýzy je prístup k hexadecimálnym a regulárnym reťazcom. Ich lexikálna analýza je riešená mierne odlišným spôsobom ako pre iné typy tokenov. Daný prístup bol zvolený s prihliadnutím na už existujúcu architektúru nástroja YARA-X a cieľ odlíšiť lexikálnu analýzu od syntaktickej. Pomerne bežným spôsobom je využitie syntaxou riadenej lexikálnej analýzy, pri ktorej parser v prípade potreby volá lexikálnu analýzu a ona mu vráti aktuálny token. Pri danom prístupe je možné aj podľa aktuálneho kontextu parsovania prispôsobovať to, akým spôsobom bude daný token reprezentovaný. Práca volí odlišný spôsob, kedy lexikálna analýza pracuje samostatne a výstupom je množina tokenov a chýb, ktorá reprezentuje daný vstup. Tá zároveň predstavuje aj vstup do samotného parsera. Pri riešení sa práca inšpiruje architektúrou nástroja `rust-analyzer`[6], ktorá rieši podobný problém.

Regulárny aj hexadecimálny reťazec sú na úrovni vstupného textu spracované ako jeden token. Tento token je definovaný pomocou regulárneho výrazu, ktorý reprezentuje všetko, čo sa v danom reťazci môže nachádzať. Následne je pri konverzii token analyzovaný druhýkrát a súčasne je aj rozdelený na podčasti, ktoré reprezentujú časti reťazca. Dôležitou poznámkou

[6] https://github.com/rust-lang/rust-analyzer/blob/master/docs/dev/architecture.md

je, že pri volení vhodnej výstupnej reprezentácie z lexikálnej analýzy sa berie ohľad na aktuálnu skladbu tokenov definovanú v pôvodnej gramatike[7]. Správne zvolenie rozdelenia už pri lexikálnej analýze výrazne zjednodušuje či už parsovanie, alebo následnú integráciu do projektu YARA-X.

Regulárne výrazy sú rozdelené na dve časti. Prvá časť je samotný literál a druhá sú jeho modifikátory. O konverziu sa stará samostatná funkcia `process_regex_string_token()`. Hexadecimálne výrazy sú spracované pomerne zložitejším spôsobom. Vo výsledku sa očakáva samostatný token pre všetky operátory alebo znaky, ktoré sú podporované ako: `'{'`, `'}'`, `'('`, `')'`, `'['`, `']'`, `'-'`, `'|'`. Ďalšími očakávanými tokenmi sú `INT_LIT`, reprezentujúci nezáporné číslo, ktoré definuje rozpätie a `HEX_LIT`. Druhé menované reprezentuje typ bajtu, ktorý sa môže nachádzať v hexadecimálnom čísle. Jedná sa o dvojicu znakov s voliteľným prefixom ~. Na oboch miesta sa môže nachádzať validný hexadecimálny znak, ktorý sa dá pomocou regulárneho výrazu definovať ako: `[0-9A-Fa-f]`. Rovnako sa na ľubovoľnom mieste môže nachádzať aj znak `'?'`, pod ktorým môžeme rozumieť akéhosi žolíka, reprezentujúceho akýkoľvek z validných hexadecimálnych znakov.

6.3.4 Syntaktická analýza

Syntaktická analýza je rozčlenená do dvoch častí. Prvá časť s názvom `syntax` sa stará o poskytnutie API naväzujúceho na knižnicu `rowan`, špecifikáciu použitej gramatiky, napojenie na parser, tvorbu abstraktného syntaktického stromu a poskytnutie typových tried pre jednotlivé uzly AST. Druhá časť má názov `parser` a jedná sa priamo o ručne písaný parser, ktorý postupne prechádza poskytnuté tokeny a vykonáva syntaktickú analýzu. Výstupom obidvoch častí je abstraktný syntaktický strom, ktorý spĺňa všetky pravidlá definované v časti 3.

Implementácia je inšpirovaná štruktúrou syntaktickej analýzy jazyka `Swift`[8] od spoločnosti Apple Inc. a architektúrou už spomínaného projektu `rust-analyzer`. Syntaktická analýza vo svojom jadre využíva knižnicu `rowan`. Ako súčasť implementácie práca poskytuje upravenú verziu tejto knižnice. V poskytnutej verzii je odstránená nadbytočná funkcionalita. Táto knižnica figuruje ako súčasť projektu `rust-analyzer` a je pod licenciou MIT. Autor tohto projektu vo videosérii *'Explaining Rust Analyzer'*[9] nabáda k jej využitiu a prípadnej úprave. Princíp fungovania knižnice `rowan`, respektíve *'Red-Green'* stromov, ktoré implementuje je vysvetlený v sekcii 5.3.2. Predstavené riešenie tento koncept rozširuje o typovanú vrstvu AST. Táto vrstva poskytuje typovaný systém pre uzly syntaktického stromu [20]. Všetky polia sú definované ako voliteľné. Je možné kedykoľvek prejsť z typovanej AST vrstvy do netypovanej syntaktickej vrstvy a naopak. Jedná sa o vrstvu, ktorá poskytuje prístupové metódy a funkcie využité parserom.

Vstupným bodom pre syntaktickú analýzu je štruktúra `SourceFile` definovaná v časti `syntax/lib.rs`. Táto štruktúra reprezentuje vstupný súbor a poskytuje nad ním metódu `parse()`. Táto metóda sa stará o získanie lexikálnych tokenov, ich spracovanie a vytvorenie výsledného abstraktného syntaktického stromu. Výsledkom je vždy syntaktický strom a množina chýb, ku ktorým došlo. Súčasťou tvorby syntaktického stromu je syntaktická analýza zhora dole. O tento prechod vstupného textu a aplikáciu gramatických pravidiel sa stará časť s názvom `parser`. Metóda `parse` na svojom vstupe očakáva inštanciu triedy

[7] https://github.com/VirusTotal/yara-x/blob/main/parser/src/parser/grammar.pest

[8] https://github.com/apple/swift/blob/13d593df6f359d0cb2fc81cfaac273297c539455/lib/Syntax/README.md

[9] https://www.youtube.com/watch?v=I3RXottNwkO&list=PLhb66M_x9UmrqXhQuIpWC5VgTdrGxMx3y

`TextTokenSource` a `TextTreeSink`. Prvá menovaná inštancia abstrahuje zdroj tokenov a poskytuje nad ním operácie ako:

- `current()` - pre získanie aktuálneho tokenu. Token nie je spotrebovaný;
- `lookahead_nth(n)` - pre získanie určitého tokenu v poradí. Poradie je definované parametrom `n`. Pokiaľ `n` = 0, výsledok je rovnaký ako pri metóde `current()`;
- `bump()` - spotrebovanie aktuálneho tokenu a presun na nasledujúci.

Každý vytvorený `Token`, ktorý je súčasťou `TextTokenSource` obsahuje svoj typ a pozíciu v texte. Biele znaky a komentáre nie sú súčasťou `TextTokenSource`.

Inštancia `TextTreeSink` sa stará o vytvorenie syntaktického stromu a abstrahuje jeho špecifické implementačné detaily. Umožňuje vytváranie uzlov, mapovanie konkrétnych tokenov na daný uzol, reprezentáciu chybových stavov, či pripájanie alebo manipuláciu s bielymi znakmi alebo komentármi. Zaujímavou časťou triedy `TextTreeSink` je implementácia metódy `n_attached_trivias()`. Táto metóda definuje nelistové uzly vo výslednom AST, ktorým môžu byť priradené komentáre. Zároveň umožňuje hierarchické usporiadanie komentárov a jednoduchšiu prácu s nimi. Je možné rozlíšiť, ktoré komentáre sa viažu ku koreňovej štruktúre, teda k celému súboru s YARA pravidlami, a ktoré ku konkrétnym pravidlám alebo ich podčastiam. Implementácia je riešená pomocou počítania bielych znakov rozdeľujúcich komentáre a jednotlivé uzly. V nasledujúcom príklade je priradený najvrchnejší komentár `//Global comment` koreňovému uzlu výsledného AST. Komentár `//Rule comment` je priradený uzlu pre pravidlo `test`, komentár `//Rule block comment` k uzlu definujúcemu telo pravidla a komentár `//String comment` k podčasti pravidla, ktoré definuje sekciu `strings`.

```
//Global comment

//Rule comment
rule test
{
    //Rule block comment

    //Strings comment
    strings:
        $a = "foo"
    condition:
        $a
}
```

Výstupom je okrem vytvoreného syntaktického stromu aj množina syntaktických chýb. Tieto chyby sú reprezentované triedou `SyntaxError`, ktorá okrem chybovej hlášky poskytuje aj pozíciu chyby vo vstupnom texte.

Súčasťou štruktúry `TextTreeSink` je štruktúra `SyntaxTreeBuilder`. Jedná sa o akýsi spojovník medzi knižnicou Rowan, ktorá je nezávislá na konkrétnom jazyku a implementácii a tvorbou AST pre konkrétny jazyk (v tomto prípade pre jazyk YARA). Súčasťou tejto triedy je definícia jazyka YARA a namapovanie všetkých tokenov na príslušné `SyntaxKind` tokeny. Takisto poskytuje aj obalenie prístupových metód knižnice Rowan a ich jednoduché volanie z kódu parsera.

Parsovanie prebieha pomocou postupného prechodu vstupných tokenov a ich validácie na základe definovaných gramatických pravidiel. Štruktúra `Parse`, ktorá je definovaná v súbore `parser/parser.rs` poskytuje základné prístupové metódy pre prácu s postupnosťou tokenov. Jedná sa prevažne o napojenie na už definovaný `TextTokenSource` a metódy pre prácu s tokenmi, ktoré poskytuje. Tieto metódy sú naviac rozšírené o:

- Porovnanie aktuálneho tokenu s očakávaným tokenom pomocou metódy `at()`;
- Overenie či je aktuálny token jeden z množiny vopred definovaných tokenov pomocou metódy `at_ts()`. Táto funkcionalita je užitočná pri zotavovaní sa z chýb, pričom sú preskakované tokeny až pokiaľ nie je aktuálny token jeden z množiny First, Follow alebo ich zjednotenia;
- Skonzumovanie určitého tokena pomocou metódy `bump()` alebo ľubovoľného tokena pomocou metódy `bump_any()`;
- Metódy pre prácu s chybami. Súčasťou tejto skupiny metód je napríklad metóda `error()` určená k vygenerovaniu chybovej hlášky a chybového stavu. Metóda `expect()` k overeniu toho, či je aktuálny token očakávaný. V prípade neúspechu je vygenerovaný chybový stav spolu s chybovou hláškou. Posledná metóda pre prácu s chybami je `err_recover()`. Táto metóda umožňuje porovnanie aktuálneho tokenu s tokenmi definovanými v synchronizačnej množine. Pokiaľ porovnanie uspeje tak vygeneruje chybovú hlášku a pokúsi sa o zotavenie. V prípade neúspechu daný token skonzumuje a vytvorí chybový uzol spolu s chybovou hláškou;

Tokeny sú priraďované jednotlivým uzlom v syntaktickom strome. Tokeny sa nachádzajú na listovej úrovni. Priraďovanie jednotlivých tokenov alebo uzlov nadradeným uzlom v strome je riešené pomocou štruktúry `Marker`. Táto štruktúra slúži k označeniu skupiny uzlov, ktoré patria k rovnakému uzlu v strome. Pri parsovaní je pomocou tejto značky vždy označený uzol, ktorý je aktuálne spracovávaný. Každý `Marker` je reprezentovaný pozíciou uzlu, ktorému patrí. Musí byť buď dokončený, alebo zahodený. Až do momentu pokiaľ nenastane jedna zo spomínaných situácii, tak sú všetky spracovávané uzly od začiatku danej značky priraďované aktuálne označenému uzlu. V prípade zahodenia aktuálneho označenia sú všetky uzly, ktoré mu doposiaľ patrili, priradené predchodcovi tohto uzlu. V prípade úspešného dokončenia označenia uzlu je označený typom `CompletedMarker`. Aj napriek ukončeniu spracovávania môže byť tento uzol ďalej rozširovaný alebo presunutý v rámci stromu pomocou metódy `precede()`. Táto funkcionalita je využitá pri precedenčnej analýze, kedy sa bežne stáva, že je potrebné už spracovaný uzol z dôvodu vyššej precedencie presunúť v rámci stromu o úroveň vyššie.

Parser produkuje sekvenciu udalostí, ktorá je napojená na štruktúru `TextTreeSink`. Existujú štyri typy udalostí:

1. **Start** - Vytvorenie uzlu.
2. **Finish** - Dokončenie uzlu. Každý uzol by mal byť buď dokončený alebo zrušený. Všetky tokeny medzi vytvorením a dokončením uzlu sú danému uzlu priradené. Využívajú sa označenia `Marker`.
3. **Token** - Vytvorenie listového uzlu, reprezentujúceho token.
4. **Error** - Vytvorenie chybového stavu vzniknutého pri parsovaní.

Implementácia samotného parsovania sa nachádza v súbore `parser/grammar.rs`. Parsovanie je implementované pomocou syntaktickej analýzy zhora nadol a to pomocou manuálneho rekurzívneho zostupu, ktorý začína v metóde `parse_source_file()`. Jediným parametrom metódy je inštancia štruktúry `Parser`, ktorá poskytuje všetky potrebné metódy. Tvorba metód, ktoré reflektujú definované gramatické pravidlá vychádza z definovanej gramatiky pre jazyk YARA[10]. Gramatika bola mierne prispôsobená a jej pozmenená verzia sa nachádza v súbore `yara.ungrammar` v koreňovom adresári kontajneru `yara-parser`. Gramatika je definovaná pomocou jazyka *Ungrammar*[11], ktorý rovnako využíva aj projekt `rust-analyzer`. Bližší popis gramatiky sa nachádza v časti 6.3.5.

Rekurzívny zostup

Parsovanie prebieha v časti `parser/grammar/`. Vstupným bodom je súbor `items.rs`, ktorý obsahuje funkciu `mod_content()`. Jedná sa o cyklus, ktorý postupne spracováva jednotlivé tokeny a prechádza gramatické pravidlá zhora nadol. Na vstupe je množina tokenov, ktorá reprezentuje celý vstupný súbor. Výstupom je vytvorenie abstraktného syntaktického stromu, kde koreňový uzol reprezentuje vstupný súbor YARA pravidiel. Postupne sa prechádza od najvyšších úrovní derivačného stromu rekurzívne k listovým úrovniam, ktoré predstavujú jednotlivé tokeny.

Súbor s YARA pravidlami môže obsahovať tri typy možných konštrukcií na najvyššej úrovni:

1. `include` výrazy - pre vloženie dodatočných YARA súborov
2. `import` výrazy - pre vkladanie podporovaných modulov
3. `rule` - samotná definícia YARA pravidiel

O spracovanie najvyššej úrovne sa stará funkcia `process_top_level()`. Pokiaľ sa pri prechode narazí na token reprezentujúci kľúčové slovo `rule` dochádza k spracovaniu tela pravidla pomocou funkcie `rule()`. Každé pravidlo môže mať voliteľné modifikátory `global` alebo `private`. Za kľúčovým slovom `rule` nasleduje identifikátor pravidla a voliteľné tagy. Telo pravidla je spracované vo funkcii `block_expression()` a je ohraničené zloženými zátvorkami. Pokiaľ dôjde k chybe na najvyššej úrovni, napríklad pri použití syntakticky nesprávnej hlavičky pravidla, je telo pravidla spracované samostatne a umiestnené pod uzol reprezentujúci chybový stav. Telo pravidla sa môže skladať z voliteľných častí `meta` a `strings`. Každá z týchto častí sa môže vyskytnúť najviac jedenkrát, ich poradie je dané a sú spracované samostatne. Sekcia `meta` je reprezentovaná dvojicou kľúč-hodnota a umožňuje užívateľom uloženie dodatočných informácií o pravidle.

Spracovanie sekcie `strings` je implementované pomocou funkcie `strings()`. Najprv sa skontroluje správnosť hlavičky sekcie a následne tela sekcie, ktoré môže obsahovať definície premenných, ktoré sú nasledované textovým reťazcom, hexadecimálnym reťazcom alebo regulárnym výrazom. Každý zo spomínaných typov je spracovaný samostatne a jednotlivé časti, z ktorých pozostáva sú po overení správnosti umiestnené do abstraktného syntaktického stromu. Nasledovať môžu voliteľné modifikátory pre daný typ reťazca. Pri výskyte chyby v tejto časti sú preskočené tokeny až po token reprezentujúci definíciu premennej. Pokiaľ sa tam takýto token nenachádza, je ďalším synchronizačným bodom sekcia `condition`.

[10] https://github.com/avast/yara-x/blob/main/parser/src/parser/grammar.pest
[11] https://github.com/rust-analyzer/ungrammar

Táto sekcia reprezentuje pravdivostný výraz, ktorého hodnota je overená a následne poskytnutá užívateľovi ako výstup YARA pravidla. Skladá sa z viacerých úrovní, pričom na každej úrovni sú podporované iné operandy a operátori. Overenie hlavičky sekcie prebieha obdobne ako pri sekcii `strings`, či `meta` pomocou funkcie `condition()`. Telo funkcie je spracované pomocou kombinácie rekurzívneho zostupu a vlastnej implementácie Pratt parsera[12]. Jednotlivé úrovne sú spracovávané pomocou implementácie Pratt parsera. Na každej úrovni je definovaná množina podporovaných operátorov spolu s číslom definujúcim úroveň precedencie a ich asociatívnosť. Pre prvú úroveň sú to operátori `and` a `or`, ktoré sú definované pomocou funkcie:

```
fn current_op(p: &mut Parser) -> (u8, SyntaxKind, Associativity) {
    match p.current() {
        T![and] => (4, T![and], Associativity::Left),
        T![or] => (2, T![or], Associativity::Left),
        _ => (0, ERROR, Associativity::Left),
    }
}
```

V prípade operátora `and` táto funkcia vracia trojicu: precedencia, typ tokenu a asociatívnosť. Precedencia je v tomto prípade **4**, čo znamená, že je vyššia ako v prípade operátori `or` a teda výraz s operátorom `and` má prioritu pred výrazom s operátorom `or`. Vo výslednom strome je tento výraz umiestnený hierarchicky nižšie ako výraz s operátorom `or`, čo znamená že je spracovaný skôr. Pre vyjadrenie hodnoty precedencie sú volené párne čísla začínajúce od 0, ktorá reprezentuje nepodporovaný operátor. Čím vyššie je dané číslo, tým je výraz obsahujúci daný operátor umiestnený hierarchicky nižšie v strome. Podporované sú či už binárne výrazy s dvomi operátormi, alebo aj unárne s jedným operátorom. Iné typy výrazov nie sú v jazyku YARA podporované.

Vyhodnocovanie pomocou Pratt parsera funguje na princípe rekurzívneho spracovania výrazov zľava doprava. Pri prvom výskyte operátora s menšou alebo rovnakou precedenciou, je rekurzívne spracovanie zastavené. Týmto spôsobom je zabezpečené správne poradie vyhodnocovania operátorov podľa ich precedencie a asociatívnosti. V prípade ľavej asociatívnosti daného operátora je k jeho hodnote precedencie pripočítaná hodnota 1. To zabezpečuje, že pri stretnutí operátorov s rovnakou prioritou sa najprv vyhodnotí ten, ktorý je vľavo. Naopak, v prípade pravej asociatívnosti sa k hodnote precedencie nič nepripočíta, a teda pri stretnutí operátorov s rovnakou prioritou sa najprv vyhodnotí ten, ktorý je vpravo.

Ako operandy v jednotlivých úrovniach výrazu môžu figurovať nie len terminálne symboly, ako napríklad premenná, pravdivostná hodnota a kľúčové slová, ale aj iné podvýrazy. Niektoré podvýrazy si vyžadujú opätovné spracovanie pomocou Pratt parsera. Každá úroveň má však inú množinou operátorov spolu s ich definovanou precedenciou. Iné podvýrazy vyžadujú len overenie postupnosti tokenov. Medzi ne patrí napríklad výraz `of`. Zaujímavé na jeho spracovaní je, že na ľavej strane výrazu od kľúčového slova `of` môže byť kvantifikátor. Podporované kvantifikátory sú: `any, all, none` alebo ľubovoľný výraz. Posledný menovaný kvantifikátor je problematický v tom, že dopredu nie je možné zistiť, aký dlhý bude daný výraz. V tomto prípade sú použité pomocné funkcie na výpočet dĺžky výrazu. Vypočítaná hodnota je následne využitá ako argument funkcie `nth()`. Táto funkcia vráti token, ktorý sa na danom mieste nachádza, čím umožňuje akési preskočenie časti textu a popredné overenie určitého tokenu. Pokiaľ nasleduje token `of` vieme, že sa jedná o podvýraz, ktorý figuruje ako kvantifikátor pre výraz `of`. Pokiaľ by sa nejednalo o výraz `of` bol

[12] https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html

by tento podvýraz spracovaný iným spôsobom a na inej úrovni abstraktného syntaktického stromu.

6.3.5 Generovanie kódu

Posledná časť implementácie parsera je vytvorenie typovanej vrstvy nad vzniknutým abstraktným syntaktickým stromom. K jej vytvoreniu dochádza v súbore `syntax/ast.rs`, ktorý obsahuje definíciu štruktúr pre uzly a tokeny (listové uzly) v danom syntaktickom strome. Tieto triedy umožňujú prevod medzi netypovanou syntaktickou vrstvou a typovanou AST vrstvou, ktorá je nad ňou postavená.

Definícia vrstvy AST sa nachádza v súboroch `syntax/ast/generated/nodes.rs` a `syntax/ast/generated/tokens.rs`. Obsah týchto súborov je automaticky vygenerovaný. Ku generovaniu kódu sa využívajú súbory v priečinku `syntax/tests/`, ktoré zároveň fungujú ako testy. V súbore `ast_src.rs` sú definované názvy pre podporované tokeny a uzly AST pre jazyk YARA. Súbor `tools.rs` obsahuje pomocné funkcie pre generovanie kódu, pridávanie hlavičiek alebo formátovanie kódu. Dôležitý je obsah posledného súbora s názvom `sourcegen_ast.rs`, ten sa stará o vytvorenie typovanej vrstvy AST. Obsahuje funkcie pre vytvorenie podporovaných tokenov typu `SyntaxKind` jazyka YARA, vygenerovanie typovanej vrstvy pre jednotlivé uzly syntaktického stromu, či funkcie k jednoduchému prevodu medzi vrstvami. K vytvoreniu danej vrstvy dochádza pomocou gramatiky `ungrammar`. Jej umiestnenie je v súbore `yara.ungram` a je inšpirované pôvodnou gramatikou pre jazyk YARA. Pri implementácii generovania kódu a využitia `ungrammar` gramatiky, práca vychádza z projektu `rust-analyzer`, ktorého autor je aj autorom `ungrammar` gramatiky. Táto gramatika narozdiel od iných gramatík neslúži na presný formálny popis parsovania, ale na popis typovanej vrstvy AST. Definuje potomkov jednotlivých uzlov a poskytuje schému, z ktorej sú následne tieto metódy generované. Jej presnejšie využitie je opísané autorom v blogu[13].

Vygenerovaný kód je rozšírený aj o časti, ktoré manuálne doplňujú funkcionalitu. Jedná sa napríklad o pravdivostné výrazy, ktoré sa môžu vyskytovať v podmienkach jazyka YARA a v gramatike `ungrammar` sú definované následovne:

```
Expression =
    BooleanExpr
|   BooleanTerm

BooleanExpr =
    lhs:Expression
        op:(
            'and'| 'or'
        )
    rhs:Expression
```

Pravdivostný výraz môže byť buď samotný operand alebo iný výraz, kde na ľavej aj na pravej strane sa nachádza iný pravdivostný výraz. Táto definícia umožňuje definovať výraz skladajúci sa z jednej alebo viacerých častí oddelených operátormi `and` a `or`. Pre jednoduchší prístup k ľavej a pravej časti výrazu je možné použiť ľubovoľné označenie. V tomto prípade je použité označenie `lhs` a `rhs` pre ľavú, respektíve pravú stranu výrazu a

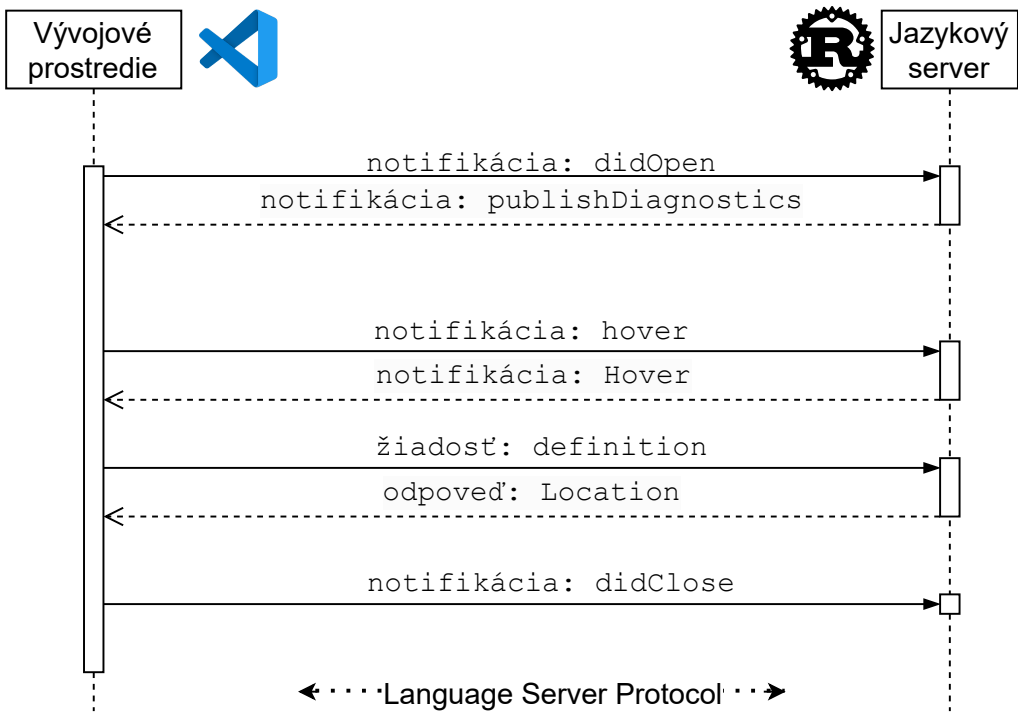
[13] https://rust-analyzer.github.io//blog/2020/10/24/introducing-ungrammar.html

označenie `op` pre operátor. Uvedené značky slúžia na oznámenie generátoru kódu, že korešpondujúce časti budú implementované manuálne. Manuálna implementácia je umiestnená v sekcii `syntax/ast/expr_ext.rs` a definuje metódy pre prístup k hodnotám operátorov či operandov jednotným spôsobom. Pri použití typovanej vrstvy AST a navigácii v strome je možné nad uzlom definujúcim `BooleanExpr` zavolať funkcie ako `op_token()`, `lhs()`, či `rhs()`, ktoré vrátia príslušného potomka daného uzlu alebo jeho hodnotu.

Bližšie použitie typovanej vrstvy, jej konverzia na syntaktickú vrstvu a navigácia v AST je znázornená vo funkcii `api_walktrough()`, ktorá sa nachádza v koreňovom adresári projektu `yara-parser` v súbore `lib.rs`. Táto funkcia slúži zároveň aj ako test.

6.4 Implementácia jazykového servera

Súčasťou vypracovaného riešenia je aj implementácia jazykového servera. Využitý je LSP[14] (**L**anguage **S**erver **P**rocol) od spoločnosti Microsoft. Jedná sa o protokol slúžiaci ku komunikácii medzi vývojovým prostredím a jazykovým serverom. Tento protokol je podporovaný väčšinou moderných vývojových prostredí [17]. Práca sa v aktuálnej podobe zameriava na jeho využitie vo vývojom prostredí VS Code[15]. Jazykový server následne poskytuje množstvo služieb využiteľných priamo vo vývojovom prostredí. Schéma 6.4 demonštruje komunikáciu medzi klientom - v tomto prípade vývojovým prostredím a jazykovým serverom.



Obr. 6.4: Komunikácia vývojového prostredia s jazykovým serverom pomocou protokolu LSP. Obrázok je inšpirovaný [23].

[14] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[15] https://code.visualstudio.com

Komunikácia servera s vývojovým prostredím prebieha pomocou troch základných typov správ: žiadostí, odpovedí a upozornení. Žiadosti reprezentujú akcie, ktoré užívateľ v danom vývojovom prostredí vykonal. Server na ne reaguje formou odpovedí a notifikácií. Notifikácie umožňujú informovať užívateľa o otvorení súboru, o jeho zavretí, či o zmenách, ktoré boli nad ním vykonané.

Vytvorenie jazykového servera je zvolené za účelom demonštrácie výhod parsera odolného voči chybám. Práca cieli na poskytnutie základných metód, ktoré uľahčia vývoj YARA pravidiel. Aktuálna implementácia poskytuje metódy pre:

- Prechod na definíciu premennej či pravidla,
- Zobrazenie výskytov premennej alebo pravidiel v súbore,
- Zobrazenie dodatočných informácii o symbole pri označení kurzorom,
- Zobrazenie hierarchie dokumentu,
- Zvýraznenie syntaktických chýb v pravidlách,
- Automatické dopĺňanie kódu pre vybrané časti pravidiel,
- Zobrazenie abstraktného syntaktického stromu,
- Zvýraznenie hierarchie pravidiel.

Implementácia poskytuje rozšírenie do vývojového prostredia Visual Studio Code, ktoré je umiestnené v priečinku `editors/vscode`. Jeho súčasťou sú konfiguračné súbory obsahujúce definíciu súborov, nad ktorými jazykový server operuje. Konfigurácia rovnako obsahuje definíciu pre syntaktické zvýrazňovanie YARA pravidiel, ktorá je prebratá z nástroja YLS. Priložený `README.md` súbor obsahuje všetky potrebné informácie k nainštalovaniu rozšírenia vrátane nastavenia cesty k implementovanému jazykovému serveru.

Zdrojové súbory k jazykovému serveru sa nachádzajú v zložke `src`. Jazykový server je implementovaný rovnako v jazyku Rust. K implementácií bola využitá knižnica `tower-lsp`[16], ktorá poskytuje asynchrónne rozhranie pre protokol jazykového servera (LSP).

Jazykový server je implementovaný v súbore `main.rs`. Dátovy typ `Backend` obsahuje definíciu komunikácie s klientom pomocou štruktúry `Client`. Rovnako obsahuje aj definíciu hashovacej tabuľky umožňujúcu súbežný prístup k obsahu zdrojového súboru či k definícií dokumentu. Reprezentácia súboru s YARA pravidlami, nad ktorými tento jazykový server operuje je popísaná pomocou štruktúry `TextDocumentItem`. Jej obsah pozostáva z definície cesty k súboru, obsahu súboru a verzie súboru.

Implementácia `LanguageServer` pre komplexný dátový typ `Backend` rozširuje jeho funkcionalitu o metódy jazykového servera. Pomocou metódy `initialize` je vykonaná inicializácia jazykového servera a sú špecifikované jeho schopnosti. Medzi tieto schopnosti patria už spomínané automatické dopĺňanie textu, zobrazenie hierarchie dokumentu či zvýrazňovanie chýb v dokumente. Implementácia týchto schopností je umiestnená vo zvyšných metódach, ktorých názov je presne stanovený knižnicou `tower-lsp`. Knižnica sa rovnako stará aj o spracovanie žiadostí od klienta a odoslanie odpovedí v správnom formáte. Úlohou jednotlivých metód je získať potrebné dáta z reprezentácie otvoreného súboru a pripojiť ich v správnom formáte k odpovedi.

[16] https://github.com/ebkalderon/tower-lsp

Metódy `did_open()` a `did_save()` obsluhujú notifikácie o otvorení, či úprave súboru. Prvá menovaná metóde volá pomocnú funkciu `on_change()`, ktorá získa aktuálnu verziu súboru a uloží ju do príslušných hashovacích tabuliek. Týmto spôsobom sa pracuje vždy s aktuálnou verziou súboru aj priamo počas jeho úpravy. Obe metódy následne využijú predstavený parser, ktorý spracuje vstupný súbor a poskytne abstraktný syntaktický strom a množinu chýb, ktoré vznikli pri spracovaní súboru. Pozícia chyby je konvertovaná do správneho formátu, ktorý LSP protokol očakáva pomocou metódy `offset_to_position()`. Následne je dvojica - chyba a jej pozícia vložená do vektoru chýb, ktorý je odoslaný naspäť klientovi pomocou metódy `publish_diagnostics()`. O spracovanie tejto odpovede sa stará priamo vývojové prostredie. Užívateľovi sú poskytnuté zobrazenia chýb vo vstupnom texte pomocou podčiarknutia príslušného rozsahu vstupu a zobrazenia chybovej hlášky. Príklad vizualizácie chybových hlášok vo vývojovom prostredí je znázornený na obrázku 6.5. Predložené pravidlo obsahuje dve syntaktické chyby. Prvá informuje o použití nesprávneho modifikátoru textového reťazca. Druhá informuje o chybnom výraze v podmienke v dôsledku zabudnutia relačného operátora.

```
rule moje_pravidlo
{
    strings:
        $textovy_retazec = "retazec" nocsae    expected a new pattern statement or pattern modifier
        $hexadecimalny_retazec = { 00 01 02 03 04 05 }
    condition:
        (
            $textovy_retazec or
            $hexadecimalny_retazec
        ) and
        filesize 100MB    invalid yara expression
}
```

Obr. 6.5: Zvýraznenie syntaktických chýb pomocou implementovaného jazykového servera vo vývojovom prostredí VS Code.

Metóda `completion()` sa stará o implementáciu podpory automatického dopĺňania kódu. Jej úlohou je rovnako získať reprezentáciu v podobe AST pomocou predstaveného parsera. Následne je táto reprezentáciu využitá k poskytnutiu množiny možných symbolov, ktoré je možné doplniť na miesto v súbore, kde sa nachádza aktuálne kurzor. Táto množina je získaná pomocou pomocnej funkcie `get_completion()`. Jej výstupom je vektor `CompletionItem` typov, ktorý je následne predaný klientovi. Tento vektor sa postupne rozširuje o podporované symboly. V aktuálnej podobe je možné poskytnúť automatické dopĺňanie pre 5 rôznych typov symbolov.

1. **`'import'` výrazy** - Je umožnené automatické dopĺňanie názvov modulov. Názvy sú dopĺňané pre riadky vo vstupnom súbore, ktoré obsahujú kľúčové slovo `'import'` nasledované úvodzovkami. Množina podporovaných modulov je získaná zo špecifikácie modulov, ktorá je priložená k zdrojovým súborom v priečinku `modules/`;

2. **modifikátory pre reťazce** - Každý z podporovaných reťazcov jazyka YARA, ktoré sú bližšie popísané v sekcii 3.1 podporuje určité modifikátory. Tieto modifikátory sú

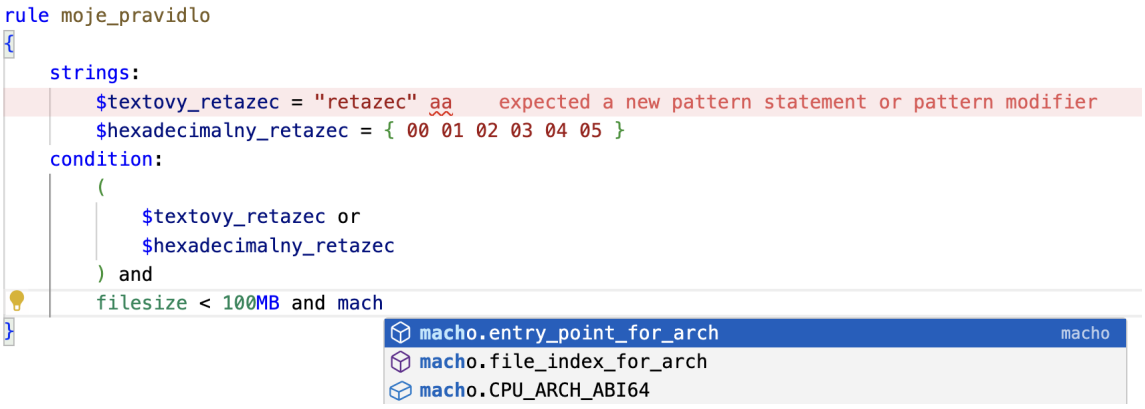
zvolené na základe typu reťazca. Dôležitým aspektom je obmedzenie ponuky automatického doplnenia modifikátorov reťazcov len v sekcii `'strings'`, v ktorej sa nachádza definícia týchto reťazcov. Toto je dosiahnuté pomocou prechodu abstraktného syntaktického stromu a porovnania aktuálnej polohy kurzoru voči jeho hierarchii;

3. **kľúčové slova v podmienkach** - Ďalším z často využívaných prvkov automatického dopĺňania sú kľúčové slová v sekcii `'condition'`. Rovnako ako pri modifikátoroch reťazcov, aj tu je najprv overená pozícia kurzoru. Pokiaľ sa vyskytuje v sekcii podmienok, tak je vytvorená množina, ktorá obsahuje všetky kľúčové slová, ktoré sa môžu na danom mieste vyskytovať. Táto množina je špecifikovaná manuálne a obsahuje kľúčové slova ako: `true`, `false`, `any` a mnoho ďalších;

4. **funkcie a konštanty z modulov** - Súčasťou jazyka YARA sú vstavané moduly. Tieto moduly rozširujú funkcionalitu jazyka a poskytujú množstvo dodatočných funkcií a konštánt, ktoré je možné následne využívať v YARA pravidlách. Ich dopĺňanie je umožnené rovnako, ako aj pri kľúčových slovách, len v sekcii `'condition'`. K získaniu tejto množiny sú využité definície modulov v priečinku `'modules'`. Jedná sa o súbory typu `JSON`, ktoré boli prevzaté z projektu Yaramod. Tieto súbory sú následne spracované a dochádza k vytvoreniu pomocných štruktúr pre každý modul. Každá štruktúra obsahuje množinu funkcií a konštánt, ktorá je prístupná pre daný modul. Tieto štruktúry sú spracované a pre každý prvok je vytvorený typ `CompletionItem`. Pri zvolení metódy alebo konštanty obsiahnutej v module, ktorý ešte nebol vložený, je vloženie tohto modulu dodatočne vložené na začiatok súboru;

5. **aktuálne podporované symboly** - Posledným typom symbolov, ktoré sú dostupné pre automatické dopĺňanie, sú aktuálne podporované symboly. Tie závisia od aktuálneho obsahu súboru s YARA pravidlami. Jedná sa o dva typy symbolov. Prvým typom sú aktuálne definované premenné v sekcii `'strings'`. Automatické doplnenie týchto premenných je následne sprístupnené v sekcii `'condition'`. Je možné využiť premenné len v rámci jedného pravidla. Pokiaľ súbor obsahuje viacero pravidiel, dostupné sú len premenné z rovnakého pravidla, v ktorého sekcii `'condition'` sa aktuálne kurzor nachádza. Druhým typom sú aktuálne definované názvy pravidiel. Každé pravidlo má svoj jedinečný názov. V sekcii `'condition'` ľubovoľného pravidla je poskytnutá množina všetkých aktuálne definovaných pravidiel. K získaniu tejto množiny, rovnako ako aj v predchádzajúcich prípadoch, je využitý AST. Vzhľadom na charakter použitého parsera, je automatické dopĺňanie aktuálne podporovaných symbolov možné aj pre pravidlá, ktorých telo nie je syntakticky správne napísané;

Obrázok 6.6 demonštruje možnosť využitia funkcií a konštánt z modulu Mach-O pri automatickom dopĺňaní v sekcii `'conditions'`.



Obr. 6.6: Automatické doplňanie funkcií a konštánt z modulu Mach-O.

Po potvrdení danej funkcie z ponuky automatického doplnenia dôjde formátovaniu funkcie do správnej podoby a presunu kurzora do tela funkcie. V prípade ak funkcia pochádza z modulu, ktorý ešte nebol vložený do zdrojového súboru s pravidlom - dôjde k jeho vloženiu. Formát súboru s pravidlom po výbere a potvrdení možnosti z automatického súboru je ukázaný na obrázku 6.7. Za povšimnutie stojí fakt, že automatické doplňanie kódu funguje bez ohľadu na syntaktickú správnosť zvyšku pravidla.

```
import "macho"
rule moje_pravidlo
{
    strings:
        $textovy_retazec = "retazec" aa    expected a new pattern statement or pattern modifier
        $hexadecimalny_retazec = { 00 01 02 03 04 05 }
    condition:
        (
            $textovy_retazec or
            $hexadecimalny_retazec
        ) and
        filesize < 100MB and macho.entry_point_for_arch()
}
```

Obr. 6.7: Vloženie modulu, ktorý obsahuje funkciu `entry_point_for_arch()`.

O zobrazenie dodatočných informácií o symbole po jeho označení kurzorom sa stará metóda `hover()`. Jej úlohou je získať dodatočné informácie o symbole a poskytnúť ich klientovi. Pomocná funkcia `get_hover()` sa snaží pre aktuálnu pozíciu kurzora získať token v AST, ktorý ho reprezentuje. Pokiaľ sa jedná o premennú, tak je zobrazená hodnota reťazca, ktorý reprezentuje. V prípade pravidla je zobrazené celé jeho telo. Výstup je poskytnutý vo formáte `Markdown`[17]. Z dôvodu odolnosti voči chybám použitého parsera je možné sa odkazovať a zobrazovať informácie aj o pravidlách, ktoré nie sú syntakticky správne. Táto funkcionalita je veľkou výhodou pri písaní pravidiel, keďže nie je podmienená syntaktickou správnosťou súboru. Príklad zobrazeného výstupu je obsiahnutý v obrázku 6.8.

[17] https://www.markdownguide.org

Toto pravidlo rovnako ako v predchádzajúcom prípade obsahuje syntaktickú chybu, ktorá je zobrazená aj vo výstupe.

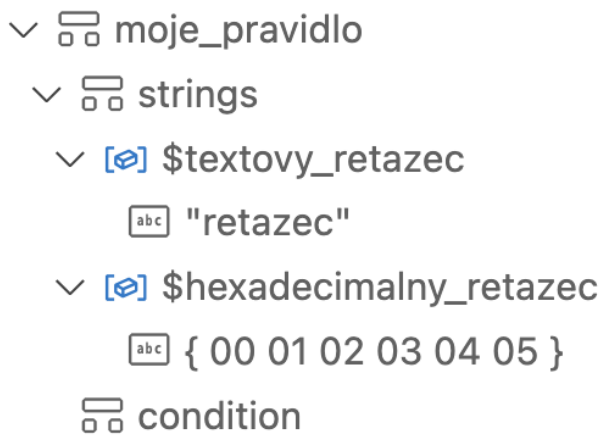
Obr. 6.8: Zobrazenie definície pravidla pri jeho označení kurzorom.

Ďalšie metódy demonštrujúce funkcionalitu použitého parsera sú `goto_definition()` a `references()`. Prvá menovaná metóda umožňuje prechod na definíciu symbolu. Pre všetky symboly, ktoré reprezentujú premenné alebo pravidlá, sa získa ich identifikátor. Následne je realizovaný prechod AST a v sekcii definície reťazcov, či definície pravidiel na globálnej úrovni je hľadaná pozícia, ktorá obsahuje definíciu symbolu s rovnakým identifikátorom. Táto pozícia je následne vrátená klientovi, ktorý umožní, aby bol na ňu presunutý kurzor. Metóda `references()` funguje obdobne akurát pre daný symbol vyhľadá všetky symboly s rovnakým identifikátorom a vráti vektor všetkých pozícií. O spracovanie tohto vektora sa stará klient a užívateľovi poskytuje ich vizuálne zobrazenie.

Metóda `code_action()` slúži na pomocný výpis AST pre aktuálne otvorený súbor. Táto metóda umožňuje v prostredí VS Code definovať akciu, ktorá zobrazí AST v konzolovom výstupe servera. Jedná sa skôr o pomocnú metódu, ktorá však môže byť užitočná pri ďalšom vývoji, ale aj pri písaní pravidiel. Pri zavolaní tejto metódy je vrátený príkaz `printAst`. Server pri zachytení ľubovoľného príkazu zavolá metódu `execute_command()`. Keďže aktuálne je podporovaný len jeden príkaz, nemusí dochádzať k dodatočnému filtrovaniu a je priamo vytvorený AST pre daný súbor a zobrazený na konzolovom výstupe. Keďže AST je vygenerovaný pre ľubovoľný vstupný súbor nezávisle na tom, koľko chýb obsahuje, je možné ho vždy zobraziť.

Poslednou implementovanou metódu je `document_symbol()`. Funkcionalita tejto metódy spočíva v zobrazení hierarchie vstupného súboru. Túto hierarchiu je následne možné zobraziť v prostredí VS Code. K zobrazeniu hierarchie je využitý postupný prechod získaného abstraktného syntaktického stromu. Na najvyššej úrovni sa môžu nachádzať výrazy typu `'import'` alebo definície pravidiel. Následne je pre každé pravidlo overený výskyt sekcií `'meta'`, `'strings'` a `'condition'`. Prvé dve menované môžu takisto obsahovať definície reťazcov alebo meta výrazov. Pre každý takýto výraz je zobrazený symbol, ktorý ho reprezentuje a typ hodnoty, ktorú drží. Výsledná štruktúra je vrátená klientovi. Okrem zobrazenia výslednej štruktúry je možné vidieť v akom mieste v hierarchií súboru sa kur-

zor nachádza. Obrázok 6.9 zobrazuje hierarchiu symbolov pre jednoduché YARA pravidlo definované v časti 3.1.



Obr. 6.9: Výsledná hierarchia pre jednoduché YARA pravidlo.

Jazykový server beží v nekonečnej slučke a obsluhuje požiadavky klienta. Okrem aktuálne definovaných metód, ktoré je schopný obslúžiť existuje aj množstvo ďalších metód. Tieto metódy je možné v budúcnosti jednoducho doplniť a jazykový server rozšíriť. Pri volení podporovaných metód práca cieli na demonštráciu výhod vytvoreného parsera. Do budúcna je možné tieto metódy rozšíriť a poskytnúť ďalšiu funkcionalitu, ktorá zjednoduší písanie YARA pravidiel.

Kapitola 7

Testovanie a vyhodnotenie

K testovaniu vytvoreného riešenia je využitá kombinácia viacerých prístupov. Jedná sa prevažne o kombináciu jednotkových testov, fuzz testov a *'end-to-end'* testov. Overenie správnosti a rýchlosti predstaveného parsera je rozšírené o porovnanie s existujúcimi alternatívami. Moduly slúžiace k testovaniu sú súčasťou implementácie jednotlivých rozšírení do YARA-X ekosystému.

Mach-O modul

Testovanie Mach-O modulu sa skladá z troch častí. Prvou sú jednotkové testy, ktoré sú priamou súčasťou implementovaného Mach-O modulu. Nachádzajú sa v podadresári `tests` a pozostávajú z viacerých funkcií, ktoré postupne kontrolujú jednotlivé funkcie. Funkcie, ktoré sa starajú o parsovanie vstupného súboru a nie sú verejne prístupné sú otestované pomocou vytvorenia vstupu bytov. Táto postupnosť bytov reprezentuje štruktúru Mach-O súboru. Môže byť či už validná, alebo nevalidná. Následne sa tieto byty vložia na vstup funkcií ako `is_macho_file_block()`, `is_32_bit()` alebo `should_swap_bytes()`. Výstup funkcií je overený voči očakávanému výstupu. Funkcie, ktoré sú prístupné aj z YARA pravidiel sú otestované iným spôsobom. Pre tento prípad je vytvorená `protobuf` štruktúra, ktorá imituje reálny výstup Mach-O modulu. Následne je využitá ako vstup do testovaných funkcií a ich výstup je obdobným spôsobom porovnaný voči očakávanému výstupu.

Druhou metódou použitou pri testovaní Mach-O modulu je využitie takzvaných 'fuzz' testov pomocou knižnice `cargo-fuzz`. Tento spôsob testovania spočíva v generovaní náhodného bytového šumu, ktorý je následne vkladaný na vstup testovaných funkcií. Zmyslom je ošetriť neočakávané správanie sa na nevalidných alebo náhodných vstupoch a zabezpečiť robustnosť implementácie. Implementácia spomínaného spôsobu testovania sa nachádza v zložke `fuzz`, ktorá je rovnako súčasťou odovzdaných súborov pre Mach-O modul. Skladá sa z dvoch zdrojových súborov, konkrétne `test_macho_file()` a `test_macho_fat_file()`. Ich úlohou je definovanie testovanej funkcie pomocou makra `fuzz_target!`. Makro sa stará o generovanie náhodného bytového šumu, ktoré je následne vkladané na vstup funkcie. Tento spôsob testovania sa predtým v YARA-X systéme nevyskytoval a bol prinesený spolu s Mach-O modulom. Okrem iného pomáha ošetriť viacero chýb v prvotnej implementácii a aktuálne je súčasťou už viacerých modulov.

Tretím zvoleným spôsobom testovania je vytvorenie testovacieho rozhrania. Toto rozhranie pôvodne vznikalo pre účely *end-to-end* testovania Mach-O modulu ako celku. V neskorších fázach bolo rozšírené a vzniklo univerzálne testovacie rozhranie pre všetky moduly nástroja YARA-X. Cieľom je otestovať reálne súbory a získať výslednú štruktúru po naparso-

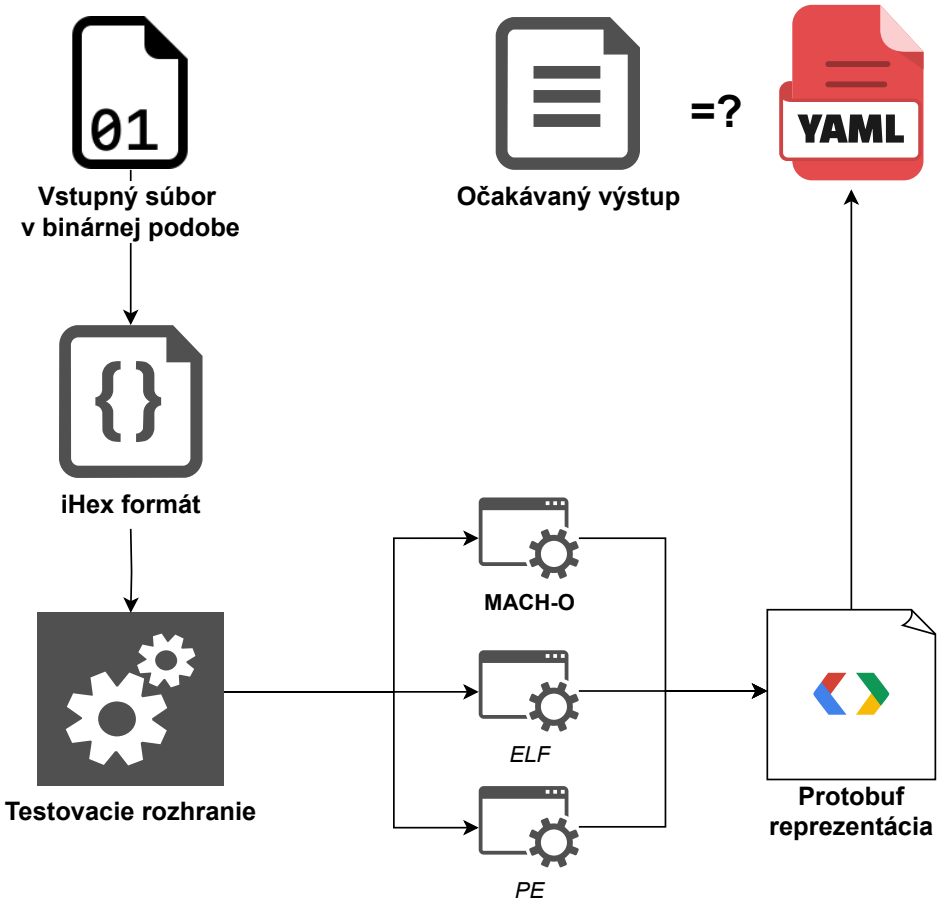
vaní jednotlivými modulmi. Keďže vstupné súbory môžu byť aj potenciálne škodlivé, je zvolený iHex[1] formát pre vstupné dáta. Tento formát umožňuje reprezentáciu binárneho súboru v textovej podobe. Testovacie rozhranie je umiestnené ako priama súčasť testov pre YARA-X projekt. Odovzdané riešenie je nasadené do hlavnej vetvy projektu a využíva sa k testovaniu modulov. Jeho princíp spočíva v prechode vstupných súborov, ich spracovaní pomocou zavolania príslušných modulov a porovnaní výstupu s očakávaným výstupom. Pre každý modul je postupne prechádzaný príslušný adresár. Tento adresár by mal okrem zdrojových súborov obsahovať aj súbory určené k testovaniu. Používa sa jednotná štruktúra, ktorá na vstupe očakáva súbory s koncovkou `'.in'` umiestnené v priečinku `tests/input`. Tieto súbory sú postupne spracované pomocou funkcie `test_modules()`. Nasleduje ich konverzia do binárneho formátu z iHex reprezentácie pomocou funkcie `create_binary_from_hex()` a zavolanie príslušného modulu pre parsovanie vstupného súboru. Získaný výstup je overený voči očakávanému výstupu. K porovnaniu výstupov je použitá knižnica `goldenfile`[2], ktorá umožňuje jednoduché porovnávanie a aktualizáciu výstupu pri zmene v implementácii.

V neposlednom rade sú doplnené aj príklady testovania pravidiel, ktoré využívajú novovzniknutý Mach-O modul na reálnych dátach. Tieto pravidlá sú doplnené do YARA-X projektu ako súčasť súboru `tests/mod.rs`.

YARA ako nástroj pre extrakciu dát

Rozšírenie ekosystému projektu YARA-X o nástroj pre extrakciu dát disponuje pomerne jednoduchou sadou testov. Jedná sa prevažne o kontrolu správneho formátu YAML výstupu. Tento formát je narozdiel od formátu JSON tvorený ručne a obohatený o prvky užívateľskej prívetivosti. Testy určené k otestovaniu jeho funkcionality sú umiestnené v súbore `tests/mod.rs` v module yara-x-proto-yaml. Princíp je podobný ako pri testovacom rozhraní, kedy je pre všetky vstupné súbory definovaný očakávaný výstup. Vstupné súbory sú vo formáte `.proto`, čo korešponduje s výstupným formátom pre moduly nástroja YARA-X. Následne je tento vstupný formát spracovaný pomocou testu `yaml_serializer()`. Získaná výstupná štruktúra dát v YAML formáte je overená voči očakávanej štruktúre. Architektúra testovacieho rozhrania je znázornená na obrázku 7.1.

[1] https://developer.arm.com/documentation/ka003292/latest/
[2] https://docs.rs/goldenfile/latest/goldenfile/



Obr. 7.1: Architektúra testovacieho rozhrania.

Dôležitým aspektom je využitie tohto modulu pri univerzálnom testovacom rozhraní, ktoré je bližšie opísané v predchádzajúcej časti. Pôvodne sa pre testovanie modulov kontroloval očakávaný výstup v protobuf formáte. Po vytvorení tohto modulu sa prešlo na kontrolu voči JSON alebo YAML formátu, ktoré obsahujú dodatočné dáta a sú následne jednoduchšie spracovateľné.

Parser odolný voči chybám

Pri testovaní novovzniknutého parsera práca využíva súbor testov. Súbor je umiestnený v priečinku `tests`, ktorý je súčasťou implementácie parsera. Nachádza sa v ňom vždy dvojica rovnomenných súborov s príponou `'.in'` alebo `'.out'`. Prvý menovaný reprezentuje vstupný súbor s YARA pravidlami, druhý menovaný očakávaný abstraktný syntaktický strom. Princíp testovania je podobný ako v predchádzajúcich prípadoch. Každý vstupný súbor je spracovaný pomocou predstaveného parsera a výsledná štruktúra abstraktného syntaktického stromu je porovnaná s jeho očakávanou štruktúrou.

Ďalším využitým spôsobom pri testovaní parsera je funkcia `api_walktrough()`. Táto funkcia predstavuje akýsi návod, ako s parserom pracovať. Obsahuje validnú aj nevalidnú reprezentáciu YARA pravidiel, pre ktoré postupne prechádza vytvorený syntaktický strom a demonštruje využitie typovanej vrstvy nad týmto stromom. Postupne ukazuje štruktúru stromu, jednotlivé uzly, dostupné funkcie a ako ich používať. Súčasťou tejto funkcie je

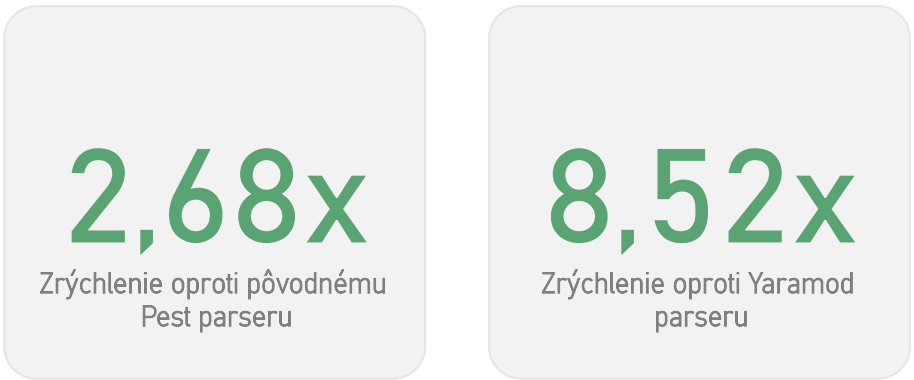
aj podrobná dokumentácia jednotlivých častí. Táto a predošlá časť sa stará o overenie správnosti na úrovni syntaktickej vrstvy.

Overenie správnosti lexikálneho spracovania je testované pomocou testov umiestnených v súbore `src/lexer/mod.rs`. Dané súbory testujú výstup z funkcie `tokenize()`, ktorá sa stará o spracovanie Logos tokenov a ich konverziu na tokeny typu `SyntaxKind`. Rovnako ako aj v predošlej časti sú otestované validné aj nevalidné vstupy.

Funkcie určené ku generovaniu kódu pre typovanú vrstvu syntaktických stromov a ku generovaniu `SyntaxKind` tokenov slúžia aj ako testy. Tieto funkcie sú označené pomocou makra `#[test]` a umožňujú opätovné pregenerovanie kódu pri zmene v gramatike, alebo v definícii tokenov.

Vytvorený parser je integrovaný aj do prostredia YARA-X, kde nahradzuje aktuálny parser. K overeniu správnosti implementácie parsera a jeho následnej integrácii je zvolené testovanie celého YARA-X ekosystému voči všetkým pravidlám definovaným v oficiálnej dokumentácii nástroja YARA [33]. YARA-X v spolupráci s novým parserom úspešne spracuje všetky definované pravidlá a úspešne nachádza zhody respektíve nezhody vo vstupných súboroch. Integrácia v aktuálnej podobe slúži na overenie správnosti riešenia a očakáva sa jej úprava pri oficiálnom nasadení. Odovzdané riešenie obsahuje aj nástroj YARA-X vrátane integrovaného predstaveného parsera. Súčasťou integrácie bolo aj overenie správnosti na už existujúcich testoch ekosystému YARA-X. Testovacia sada bola upravená aby reflektovala odovzdanú verziu nástroja. Popis ako spustiť testy pre implementované rozšírenia a testy ekosystému YARA-X je priložený v dokumente `README.md`.

Okrem overenia správnosti implementovaného parsera je porovnaná aj výkonnosť a efektivita daného riešenia. Dosiahnuté výsledky na obrázku 7.2 pri použití predstavenej verzie parsera dosahujú priemerne 2.67 násobné zrýchlenie oproti použitiu pôvodného parsera a viac ako 8 násobné zrýchlenie oproti použitiu alternatívneho parsera Yaramod. Všetky nasledujúce namerané výsledky boli dosiahnuté na verejnej sade YARA pravidiel[3] a je možné ich jednoducho zreprodukovať. Dataset obsahuje 510 súborov s pravidlami. Každé pravidlo bolo spustené 1000x predstaveným parserom, pôvodným parserom nástroja YARA-X a Yaramod parserom. Získaný bol priemer spracovania daného súbora pre každý parser. Všetky tri nástroje úspešne spracovali všetky súbory s pravidlami a vytvorili abstraktný syntaktický strom.

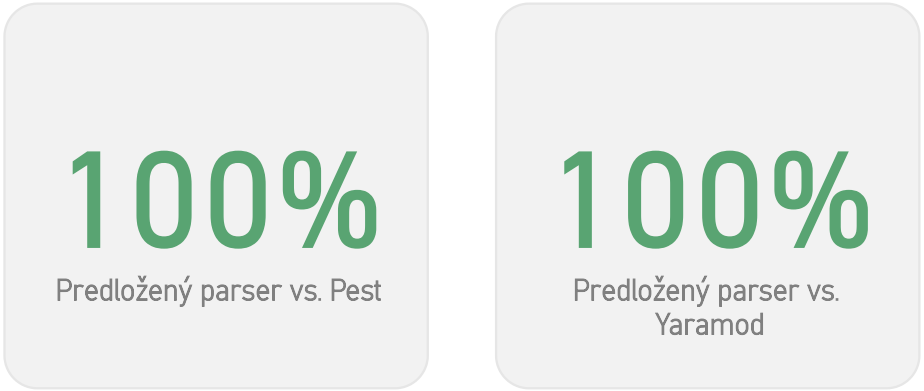


Obr. 7.2: Priemerné zrýchlenie predstaveného parsera voči alternatívam.

[3] https://github.com/Yara-Rules/rules

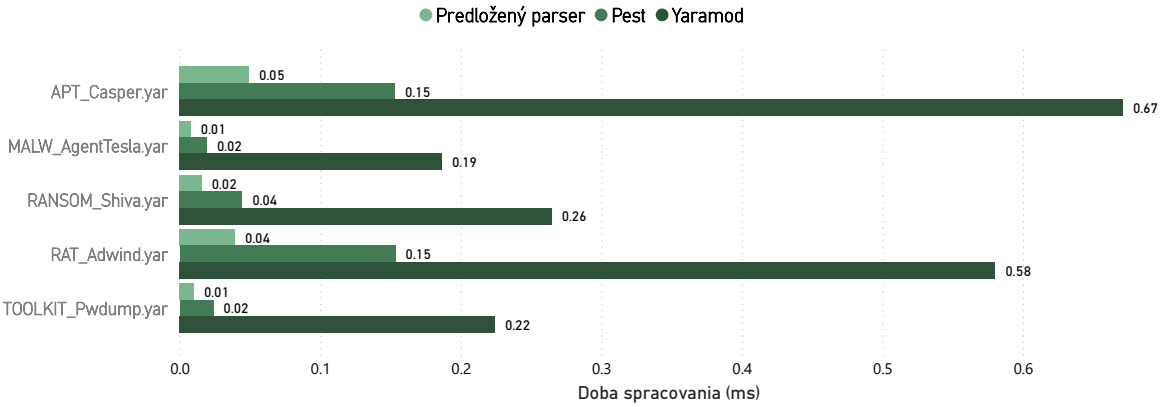
Dosiahnuté zrýchlenie má za následok použitie ručne-písaného parsera, ktorý dokáže efektívne pracovať so vstupným textom. Významný vplyv na zrýchlenie mohlo dosiahnuť aj oddelenie sémantickej kontroly. Predstavený parser sa stará len o lexikálnu a sémantickú analýzu pričom pôvodný parser v sebe obsahoval aj prvky sémantickej kontroly. Sémantická kontrola je ponechaná čisto na kompilátor nástroja YARA-X. Yaramod síce umožňuje oddialiť niektoré sémantické kontroly, ale na výsledný čas to nemá vplyv, keďže skôr či neskôr ku ním dôjde.

Pri spracovaní všetkých YARA pravidiel z testovacej sady bol rýchlejší predstavený parser. Testovacia sada obsahuje množstvo súborov o rôznej veľkosti. Obrázok 7.3 demonštruje pomer súborov, pre ktoré predstavený parser spracoval za kratší čas.



Obr. 7.3: Pomer súborov, na ktorých bol predstavený parser rýchlejší voči alternatívam.

Graf 7.4 demonštruje dobu behu parserov nad vybranými súbormi. Bolo zvolených päť súborov ako zástupcov každej kategórie malvéru pre daný testovací dataset. Z grafu vyplýva, že vo všetkých prípadoch bol značne rýchlejší predstavený parser.



Obr. 7.4: Priemer doby behov parserov nad vybranými súbormi.

Graf vychádza z tabuľky 7.5, v ktorá obsahuje dobu behu parserov nad vybranými súbormi v milisekundách. V tabuľke je vizuálne naznačené porovnanie s predstaveným parserom

Názov súboru	Predložený parser	Pest	Yaramod
APT_Casper.yar	0.05	0.15	0.67
MALW_AgentTesla.yar	0.01	0.02	0.19
RANSOM_Shiva.yar	0.02	0.04	0.26
RAT_Adwind.yar	0.04	0.15	0.58
TOOLKIT_Pwdump.yar	0.01	0.02	0.22

Obr. 7.5: Porovnanie doby behu parserov nad vybranými súbormi (v ms).

Okrem testovania na verejne dostupnom datasete prebehlo aj testovanie nad interným datasetom firmy Gen Digital Inc., ktorý obsahuje všetky pravidlá používané v produkcii. Všetky pravidla boli úspešne spracované a bol získaný výsledný AST.

Vzhľadom na dosiahnuté výsledky práce bol autor pridaný medzi oficiálnych prispievateľov[4] projektu YARA-X a firma Gen Digital Inc., ktorú reprezentuje bola pridaná ako spoluautor projektu a vlastník práv[5].

[4] https://github.com/VirusTotal/yara-x/blob/main/CONTRIBUTORS
[5] https://github.com/VirusTotal/yara-x/blob/main/AUTHORS

Kapitola 8

Záver

Cieľom práce je rozšírenie ekosystému nástroja YARA-X a umožnenie jeho následného jednoduchého využitie pre potreby statickej analýzy súborov pomocou YARA pravidiel, interaktívne vyhodnocovanie YARA pravidiel a využitie v integrovaných vývojových prostrediach. Práca skúma aktuálnu verziu nástroja YARA a jej nedostatky v porovnaní s novovznikajúcou verziou nazývanou YARA-X, ktorá je vyvíjaná v jazyku Rust. Riešenie je vypracované v spolupráci s firmou Gen Digital, ktorá je zadávateľom práce a je taktiež konzultované s autorom nástroja YARA, Victorom M. Alvarezom.

Práca predkladá rozšírenie nástroja YARA-X o Mach-O modul, ktorý umožňuje písanie pravidiel pre potreby statickej analýzy súborov určených pre operačný systém MacOS. Tento modul je schválený a nasadený do hlavnej vetvy projektu, kde je dostupný všetkým užívateľom. Súčasťou riešenia je aj rozšírenie ekosystému o nástroj umožňujúci jednoduché zobrazenie informácií získaných z podporovaných modulov. Tento nástroj umožňuje malvérovým analytikom získať prehľad o skúmanom súbore ešte predtým, ako začnú s písaním YARA pravidiel. Odstraňuje nutnosť využívať iné externé nástroje pri prvotnej analýze súboru, ktoré častokrát prinášajú nepresnosti a takisto odstraňuje problémy s tým, že ich výstup nie je vždy kompatibilný s nástrojom YARA. Zobrazené informácie sú v strojovočitateľnej podobe a zároveň vizuálne prívetivé. Tento nástroj je takisto schválený a nasadený do hlavnej vetvy projektu. Posledným vypracovaným rozšírením je parser odolný voči chybám. Je vytvorený jednotný parser, ktorý ponúka možnosť zotavenia sa z chýb. Ďalšou výhodou predstaveného rozšírenia je možnosť jednoduchej integrácie externých nástrojov do ekosystému YARA-X. Jedno z možných využití predstaveného parsera je v práci demonštrované na vytvorení jazykového servera, ktorý uľahčuje užívateľom prácu pri vytváraní nových YARA pravidiel.

Práca bola riadne otestovaná a riešenie disponuje viacerými sadami testov. Súčasťou riešenia je vlastné testovacie rozhranie, ktoré umožňuje pomocou sady vstupov a k ním korešpondujúcim výstupom univerzálne otestovať všetky podporované moduly. Výsledné riešenie využíva rozšírenie nástroja YARA-X o nástroj pre extrakciu dát zo súborov. Testovacia sada je doplnená o jednotkové testy jednotlivých funkcií, zdrojových súborov a o *fuzz* testy, ktoré umožňujú testovanie odolnosti kódov pomocou generovania náhodného bytového šumu. Na záver je predstavený parser porovnaný s existujúcimi alternatívami. Je možné konštatovať, že práca dosiahla takmer trojnásobné zrýchlenie oproti pôvodnému parseru a viac ako osemnásobné zrýchlenie oproti doteraz používanej alternatíve.

Práca sa svojimi výsledkami zaslúžila o pridanie jej autora a firmy, ktorú reprezentuje, medzi spoluvlastníkov projektu spolu s firmou Google Inc., ktorá projekt YARA-X

zastrešuje. V čase dokončovania práce sa čaká na prvé oficiálne vydanie nástroja YARA-X alternatívy a postupne aj náhrady nástroja YARA.

Autor plánuje v práci pokračovať. Medzi plánované rozšírenia do budúcnosti patrí finálna integrácia predstaveného parsera do projektu YARA-X. Parser je možné rozšíriť o ďalšie aspekty, akými sú podpora spracovania viacerých súborov, pridanie ďalšej funkcionality do jazykového servera, či vytvorenie nových rozšírení, ktoré budú aktuálny parser využívať. Medzi takéto rozšírenia môže patriť napríklad tvorba YARA pravidiel z kódu, či automatická oprava syntakticky nesprávnych pravidiel.

Literatúra

[1] Alvarez, V. M. *Module developer's guide*. Dec 2023 [cit. 2024-01-27]. Dostupné z: https://github.com/VirusTotal/yara-x/blob/main/docs/Module%20Developer's%20Guide.md.

[2] Andriesse, D. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. No starch press, 2018. ISBN 9781593279127.

[3] Apple. *Overview of the Mach-O Executable Format*. Mar 2014. Dostupné z: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html.

[4] Bayer, U., Moser, A., Kruegel, C. a Kirda, E. Dynamic analysis of malicious code. *Journal in Computer Virology*. Springer. 2006, zv. 2, s. 67–77.

[5] Belcic, I. *What Is Malware and How to Protect Against Malware Attacks?* Avast, Oct 2023 [cit. 2023-10-28]. Dostupné z: https://www.avast.com/c-malware.

[6] Bishonen, N. *Rust users team Samara meetup*. Apr 2020 [cit. 2024-01-04]. Dostupné z: https://gist.github.com/humb1t/086cbba82ef3220f08d9899ce1f1deca.

[7] Bybydev. *Why rust has such a complex syntax*. 2023 [cit. 2023-12-11]. Dostupné z: https://byby.dev/rust-complex-syntax.

[8] Committee, T. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. May 1995 [cit. 2023-11-07]. Dostupné z: https://refspecs.linuxbase.org/elf/elf.pdf.

[9] Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Hruška, T. et al. Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. In: Kim, T.-h., Adeli, H., Robles, R. J. a Balitanas, M., ed. *Information Security and Assurance*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 72–86. ISBN 978-3-642-23141-4.

[10] Ďuriš, T. *Know your yara rules series: 4 Yara-X: The next generation*. Nov 2023 [cit. 2023-12-14]. Dostupné z: https://engineering.avast.io/know-your-yara-rules-series-4-yara-x-the-next-generation/.

[11] Egele, M., Scholte, T., Kirda, E. a Kruegel, C. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computing Surveys - CSUR*. Február 2012, zv. 44, č. 02, s. 1–42. DOI: 10.1145/2089125.2089126.

[12] Gandotra, E., Bansal, D. a Sofat, S. Malware Analysis and Classification: A Survey. *Journal of Information Security*. Január 2014, zv. 05, s. 56–64. DOI: 10.4236/jis.2014.52006.

[13] Idika, N. a Mathur, A. A survey of malware detection techniques. *Purdue University*. Marec 2007.

[14] Jeyashankar, A. *Most Common Malware Obfuscation Techniques*. Feb 2022 [cit. 2023-11-01]. Dostupné z: https://www.socinvestigation.com/most-common-malware-obfuscation-techniques/.

[15] Jung, R. *Understanding and evolving the Rust programming language*. 2020. Dizertačná práca. Saarland University.

[16] Jung, R., Jourdan, J.-H., Krebbers, R. a Dreyer, D. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2017, zv. 2, POPL, s. 1–34.

[17] Kaštán, M. *Vývojové prostředí pro jazyk YARA*. 2021. Diplomová práca. Vysoké učení technické v Brně, Fakulta informačních technologií.

[18] Kašťák, M. *Yari: A new era of Yara Debugging*. Sep 2022 [cit. 2023-12-01]. Dostupné z: https://engineering.avast.io/yari-a-new-era-of-yara-debugging/.

[19] Kašťák, M. *YLS: First Step towards yara development environment*. Aug 2022 [cit. 2023-12-01]. Dostupné z: https://engineering.avast.io/yls-first-step-towards-yara-development-environment/.

[20] Kladov, A. *Syntax in rust-analyzer*. Jan 2024 [cit. 2024-04-26]. Dostupné z: https://github.com/rust-lang/rust-analyzer/blob/master/docs/dev/syntax.md.

[21] Microsoft. *A proactive approach to more secure code*. 2019 [cit. 2023-12-02]. Dostupné z: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[22] Microsoft. *PE Format*. Mar 2023 [cit. 2023-11-07]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format.

[23] Microsoft. *Language Server Protocol*. Dec 2024 [cit. 2024-04-06]. Dostupné z: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/.

[24] Naik, N., Jenkins, P., Savage, N., Yang, L., Boongoen, T. et al. Embedded YARA rules: strengthening YARA rules utilising fuzzy hashing and fuzzy rules for malware analysis. *Complex & Intelligent Systems*. Springer. 2021, zv. 7, s. 687–702.

[25] Naik, N., Jenkins, P., Savage, N., Yang, L., Naik, K. et al. Embedding Fuzzy Rules with YARA Rules for Performance Optimisation of Malware Analysis. In: *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. 2020, s. 1–7. DOI: 10.1109/FUZZ48607.2020.9177856. ISBN 9781728169323.

[26] Osenkov, K. *Roslyn Immutable Trees*. Feb 2020 [cit. 2024-04-26]. Dostupné z: https://github.com/KirillOsenkov/Bliki/wiki/Roslyn-Immutable-Trees.

[27] Rad, B. B., Masrom, M. a Ibrahim, S. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*. 2012, zv. 12, č. 8, s. 74–83.

[28] Roth, F. *Yara Performance Guidelines*. Feb 2021 [cit. 2023-11-28]. Dostupné z: https://gist.github.com/Neo23x0/e3d4e316d7441d9143c7.

[29] Saeed, I., Selamat, A. a Abdelrahman, A. A Survey on Malwares and Malware Detection Systems. *International Journal of Computer Applications (IJCA)*. Apríl 2013, Vol. 67, s. pp. 25–31.

[30] Security, U. H. *ICS-CERT monitor - CISA*. May 2015 [cit. 2023-11-28]. Dostupné z: https://www.cisa.gov/sites/default/files/Monitors/ICS-CERT_Monitor_May-Jun2015.pdf.

[31] Symantec. *Internet Security Threat Report*. Symantec, Apr 2016 [cit. 2023-10-29]. Dostupné z: https://docs.broadcom.com/doc/istr-21-2016-appendices-en.

[32] Thomas, R. *Lief: Library to instrument executable formats*. Quarkslab, 2017 [cit. 2023-11-07]. Dostupné z: https://www.romainthomas.fr/publication/slides/17-07-RMLL-LIEF.pdf.

[33] VirusTotal. *Welcome to YARA's documentation!* 2014 [cit. 2023-11-27]. Dostupné z: https://yara.readthedocs.io/en/stable/index.html.

[34] Webroot. *2018 Webroot Threat Report*. Webroot, 2018 [cit. 2023-11-02]. Dostupné z: https://www-cdn.webroot.com/9315/2354/6488/2018-Webroot-Threat-Report_US-ONLINE.pdf.

[35] Wolf, A. *Most common malware attacks*. Oct 2023 [cit. 2023-10-28]. Dostupné z: https://arcticwolf.com/resources/blog/8-types-of-malware/.

[36] Ye, Y., Li, T., Adjeroh, D. a Iyengar, S. A Survey on Malware Detection Using Data Mining Techniques. *ACM Computing Surveys*. 03. vyd. Jún 2017, zv. 50, č. 41, s. 1–40. DOI: 10.1145/3073559.

[37] Yin, H., Song, D., Egele, M., Kruegel, C. a Kirda, E. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In: Computing Machinery, A. for, ed. *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2007, s. 116–127. CCS '07. DOI: 10.1145/1315245.1315261. ISBN 9781595937032. Dostupné z: https://doi.org/10.1145/1315245.1315261.