VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠIŘOVÁNÍ JAZYKA YARA

EXTENDING YARA LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ KENDER**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kender Tomáš**
Program: Informační technologie
Název: **Rozšiřování jazyka YARA**
Extending YARA Language
Kategorie: Bezpečnost

Zadání:

1. Prostudujte si jazyk pro popis vzorů, YARA (https://github.com/VirusTotal/yara) a jeho moduly. Soustřeďte se i na vnitřní detaily jako implementace parseru, lexeru nebo reprezentace bajtkódu.
2. Seznamte se s tím, jak se jazyk YARA používá ve společnosti Avast pro detekci škodlivých souborů.
3. Navrhněte vylepšení jazyka YARA, která vylepší jazyk YARA o nové syntaktické konstrukty, které umožní malwarovým analytikům snazší zápis YARA pravidel. Navrhněte taktéž vylepšení YARA modulů o nové funkce.
4. Po konzultaci s vedoucím implementujte navržená vylepšení navrhnutá v bodě 3 a otestujte je na skutečných vzorcích malwaru.
5. Svoji práci zhodnoťte a uveďte výhled na další možný vývoj.

Literatura:

- Dokumentace jazyka YARA (https://yara.readthedocs.io/en/latest/)
- Levine, John R. Flex & Bison. 1st ed., O'Reilly Media, 2009.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních tří bodů a rozpracování bodu čtvrtého.

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

Vedoucí práce: **Regéciová Dominika, Ing.**
Konzultant: Milkovič Marek, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 22. října 2020

Abstrakt

Táto práca sa zaoberá problematikou vylepšení nástroja YARA slúžiaceho na definovanie vzorového chovania malvéru a následné vyhľadávanie na základe definovaných vlastností v súboroch za účelom detekcie malvéru v prehľadávaných súboroch. Zavádza nové syntaktické konštrukty jazyka na zápis vlastností, načrtáva nový spôsob vyhľadávania reťazca v behaviorálnych informáciach generovaných Cuckoo Sandboxom a vyhodnocuje dopady zmien. Pri riešení budeme pracovať so zápisom lexikálnych a syntaktických pravidiel jazyka, pridáme do YARY nový dátový typ pre dynamické pole, ale budeme sa venovať aj optimalizácii výkonu bajtkódu či implementácii nového bajtkódového príkazu. Výstupom práce je produkt, ktorý umožní malvérovým analytikom písať kratšie a ľahšie čitateľné pravidlá pre detekciu malvérov a skrátiť dobu skenovania behaviorálnych informácií.

Abstract

This thesis is focused at improvements for a tool called YARA, which is used for describing malware patterns and finding these patterns in files that are subject for scanning. We will add new syntactic features and improve the scanning process of behavioral files generated by Cuckoo Sandbox. During the process of adding these features, we will extend lexical and syntactic rules of the language, introduce a dynamic array type, optimize bytecode and implement a new command for it. The output of this thesis is going to be a new version of YARA that simplifies rule writing for malware analysts and aims to improve scanning performance of behavioral data.

Kľúčové slová

YARA, kompilátor, Flex, Bison, bajtkód, assembler, behaviorálna analýza, Hyperscan

Keywords

YARA, compiler, Flex, Bison, bytecode, assembler, behavioral analysis, Hyperscan

Citácia

KENDER, Tomáš. *Rozšiřování jazyka YARA*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Dominika Regéciová

Rozšiřování jazyka YARA

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Dominiky Regéciovej a konzultanta Ing. Mareka Milkoviča. Uviedol som všetky literárne zdroje, publikácie a dalšie zdroje, z ktorých som čerpal.

.....
Tomáš Kender
11. mája 2021

Poďakovanie

Touto cestou by som chcel vysloviť poďakovanie všetkým ľuďom, ktorí ma sprevádzali pri písaní tejto práce, počnúc mojou vedúcou Ing. Dominikou Regéciovou a externým konzultantom Ing. Marekom Milkovičom za nespočetné rady a nápady, ale aj mojim priateľom a rodine za podporu v ťažkých chvíľach.

Obsah

Kapitola 1

Úvod

Žijeme v dobe, kedy digitalizácia prebieha rýchlejšie než kedykoľvek pred tým v histórii. Do virtuálneho sveta sa presúva komunikácia, zábava i biznis. Toto všetko len ďalej umocňuje aktuálna situácia vo svete prameniaca z víru COVID-19, vďaka ktorému sa po prvý raz v histórii presunuli na počítačovú obrazovku i tradičné domény reálneho sveta, ako výuka na verejných školách alebo masívny a v princípe bezprecedentný presun pracovnej sily z pracovísk na prácu z domu. Aj tieto faktory prispeli k tomu, že za zhruba 10 dní lockdownu stúplo vyťaženie internetu až o 15-20% [3].

Rozmachom internetu sme sa dostali do bodu, kedy sa začala elektronická populácia sveta nekontrolovateľne navyšovať a tento trend, zdá sa, bude len ďalej akcelerovať. Do roku 2025 sa očakáva, že by mohol narásť počet zariadení Internet of Things zo súčasného odhadu 20.4 biliónov až na zhruba 75 biliónov, čo je nárast na 367% počtu súčasných zariadení [8].

Z prudkého odklonu od konvenčných komunikačných metód k internetu sa tak stala živná pôda pre internetových zločincov, ktorí v novej technológii zacítili priestor pre zárobok. Zatiaľ čo tento rok by mali škody spôsobené kybernetickými útokmi dosiahnuť 6 triliónov dolárov, do roku 2025 by to mohlo byť až 10.5 triliónov [6].

Na druhej strane pomyselnej barikády stoja malvéroví analytici. Ich úlohou je identifikovať nové hrozby a nájsť spôsob, ako zamedziť ich ďalšiemu šíreniu. Pri tomto boji využívajú širokú škálu technológií, okrem iného i nástroj YARA. Tú si za posledné roky obľúbilo veľké množstvo organizácií, ktoré si ju osvojili a dennodenne ňou pomáhajú odhaľovať malvéry skryté pod rúškom nevinne vyzerajúcich programov, ktoré by vás ani nenapadlo upodozrievať zo záškodníckych aktivít.

YARE aj napriek jej robustnosti naďalej chýbajú mnohé vymoženosti, ktoré by analytikom zjednodušili každodennú prácu. V kapitole 2 si vysvetlíme, čo presne robí nástroj YARA, ako sa používa, z akých fáz sa skladá jeho beh, ako sa v ňom zapisujú gramatické pravidlá jazyka a ako prebieha skenovanie, v kapitole 3 navrhneme niekoľko úprav pre komfortnejšie používanie nástroja. Pôjde o pridanie konceptu interných premenných pravidiel, pridanie operátora `of`, ale aj zmena spôsobu skenovania behaviorálnych dát. Tu si rozoberieme motiváciu za navrhovanými zmenami, ale aj teoretické podklady potrebné k pochopeniu problematiky. V kapitole 4 tieto zmeny implementujeme a použijeme pri nich aj ďalšie techniky, ako napríklad pole s dynamickou dĺžkou a zníženie počtu presunov dát v pamäti. V kapitole 6 nás čaká vyhodnotenie odvedenej práce a ďalšie plány do budúcna.

Kapitola 2

YARA ako jazyk a nástroje na ňom založené

YARA je open-source nástroj vyvinutý firmou VirusTotal, ktorý sa používa na identifikáciu a klasifikáciu malvéru na základe zápisu jeho vlastností. Na to používa súbory obsahujúce popisy jednotlivých malvérov, ktoré sa v prehľadávaných súboroch môžu nachádzať a poskytnuté súbory na skenovanie. Každý popisný súbor sa skladá z jednotlivých pravidiel (rules), ktoré popisujú či už kódovú charakteristiku malvéru, alebo jeho aktivitu.

U prvej varianty ide o zápis hľadaných reťazcov či regulárnych výrazov, ktoré sa následne hľadajú v skenovaných súboroch. Prostredníctvom modulov sa dá pracovať i s hlavičkami súborových formátov, napríklad PE [5] a ELF [7].

V prípade aktivity programu máme na mysli jeho dynamické behaviorálne vlastnosti zapísané do behaviorálnych reportov. V nich sú zachytené napríklad názvy mutexov, ktoré program vytvára, alebo sieťové adresy, súbory a registre, ku ktorým program pristupuje. Analyzované vlastnosti sa hľadajú vo výstupe nástroja Cuckoo Sandbox[1].

Tieto vlastnosti môžeme v pravidlách vzájomne kombinovať tak, aby sme na jednej strane dokázali zachytiť čo najväčší počet mutácií toho istého malvéru, ale zároveň nezachytávali súbory, ktoré sú malvéru len podobné a v skutočnosti malvérom nie sú. V skutočnom svete sa ale k takémuto perfektnému stavu nedostaneme, a preto môžeme hovoriť vždy len o snahe minimalizovať počet `false positives` (falošných hlásení prítomnosti malvéru) a `false negatives` (nehlásení prítomnosti malvéru v súboroch, ktoré daný malvér obsahujú).

2.1 Zápis pravidla a jeho spustenie nad súborom

Minimalistické pravidlo zapísané syntaxou jazyka YARA:

```
rule moje_pravidlo
{
    condition:
        false
}
```

Toto pravidlo obsahuje najmenšie množstvo prvkov, aké môže: názov pravidla a definíciu jednoduchej podmienky, na základe ktorej pravdivostnej hodnoty skener rozhodne,

[1] https://cuckoosandbox.org/

či analyzovaný súbor koreluje s pravidlom a teda obsahuje popisovaný malvér. V tomto prípade samozrejme nedochádza k žiadnej reálnej analýze súboru - pre žiadny súbor sa pravidlo nevyhodnotí kladne, keďže vždy vracia zápornú pravdivostnú hodnotu bez toho, aby niečo hľadal v potenciálne malvérovom súbori.

Podrobný popis zostavenia a spustenia YARY je dostupný v oficiálnej dokumentácii [9]. Pre nás je podstatné len zostavenie programu s aktivovaným Cuckoo modulom a jeho následné spustenie nad prehľadávaným súborom s nami definovaným pravidlom:

```
./bootstrap.sh
./configure --with-cuckoo-module
./build.sh
./yara moje_pravidlo.yar prehladavany_subor
```

Pri vývoji sa okrem toho využívali pri volaní skriptu `configure` aj prepínače `--enable-debug` a `--disable-optimization`.

2.2 Reťazce a regulárne výrazy

Pre to, aby sme mohli podrobne popísať malvér, potrebujeme viac konštrukcií, než len `false`. Potrebujeme konštrukcie, ktorými budeme môcť prehľadávať súbory. Z tohto dôvodu si musíme zaviesť spôsob definovania:

- textových reťazcov,
- hexadecimálnych reťazcov,
- regulárnych výrazov.

Všetky reťazce, ktoré máme záujem hľadať v skenovaných súboroch, zapisujeme do samostatnej položky reťazce (strings) pod svojim unikátnym identifikátorom a zvnútra podmienky už len referencujeme priamo identifikátory týchto reťazcov. Platí pravidlo, že každý identifikátor musí začínať dolárom, napríklad `$moj_retazec`.

```
rule moje_pravidlo
{
    strings:
        $text = "Ahoj Svete!"
        $hexa = { D1 28 C8 AA }
        $regexp = /ABCD(2|4).5[A-F]{10}/
    condition:
        1 of ($text,$hexa,$regexp)
}
```

Pri definovaní reťazcov je možné používať aj modifikátory, napríklad `fullword` (pred a za zhodou sa nenachádza alfanumerické písmeno) alebo `nocase` (pri hľadaní zhody sa nerozlišujú písmená veľkej a malej abecedy). Viac príkladov modifikátorov a ich použitia nájdete v oficiálnej dokumentácii[2].

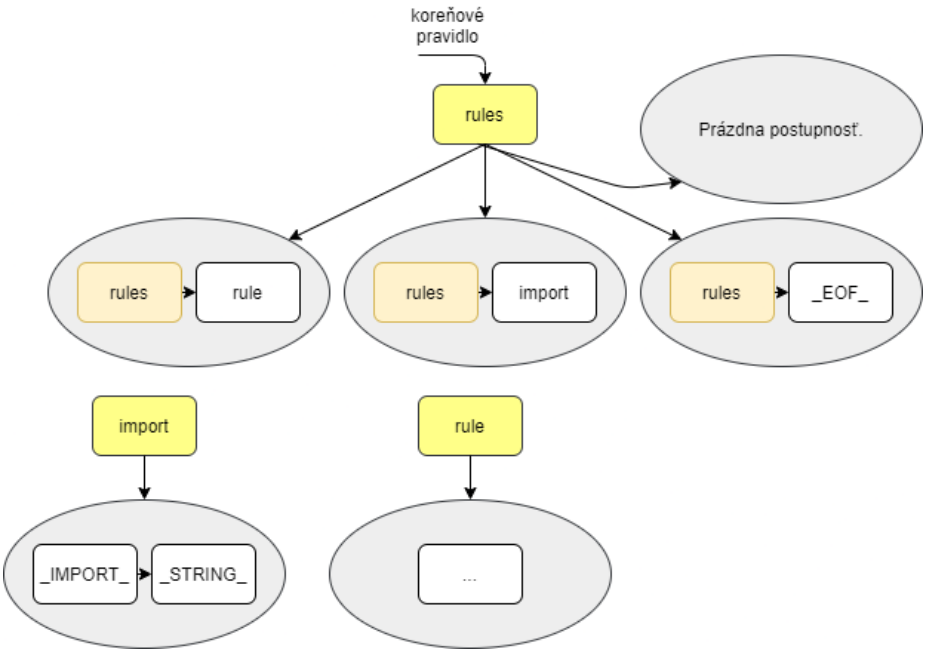
[2]Popis modifikátorov je dostupný na adrese yara.readthedocs.io/en/stable/writingrules.html#string-modifier-summary

V minulosti sa používala pre regulárne výrazy knižnica určená pre PCRE (Perl Compatible Regular Expressions[3]) formát regulárnych výrazov. Dnes už YARA využíva vlastnú implementáciu spracovania výrazov. Podporuje väčšinu funkcionality PCRE formátu až na výnimky, ako napríklad záchytné skupiny (capture groups), spätné referencie (backreferences) alebo podmienené výrazy.

Pri referencovaní reťazcov v podmienke ale nemusíme pracovať výlučne len s informáciou, či sa daná správa nachádza v súboroch, alebo nie. Môžeme tiež hľadať zhodu na konkrétnom mieste v súbore (`$str at 100`), v rozmedzí medzi dvomi bodmi v súbore (`$str in (0..100)`) prípadne môžeme hľadať len súbory s konkrétnym počtom nájdených reťazcov (`#str > 3`).

2.3 Skenovanie pravidiel

Lexikálne a syntaktické pravidlá jazyka YARA sú zapísané prostredníctvom nástroja Flex a Bison. Tie sú zapísané v súbore `grammar.y`, ktorý sa pri preklade preloží do kódu v jazyku C a následne sa preloží do strojového kódu spolu s ostatnými zdrojovými kódmi. Gramatické pravidlá YARY sú zapísané formou syntaktického stromu, kde sa každá časť pravidla dá popísať ako postupnosť iných logických celkov a tokenov.



Obr. 2.1: Vizualizácia logickej štruktúry pravidiel.

Celá gramatika je pritom založená na tzv. `koreňovom pravidle` (pravidlo, v ktorom začína a v tomto prípade aj končí spracovanie súboru). Keďže jeden súbor môže obsahovať ľubovoľný konečný počet pravidiel, nachádza tu využitie ďalší dôležitý prvok z prekladačovej praxe, `rekurzíva` pravidla. Ako vidíme na obrázku 2.1, niektoré vetvy koreňového pravidla obsahujú na prvom mieste postupnosti znova to isté koreňové pravidlo. To prekladaču umožňuje pri prechádzaní súborom postupne sa vnorovať hlbšie a hlbšie do toho istého pravidla, až sa vnorí do tej vetvy pravidla, ktorá už ďalšiu rekurziu neobsahuje (v tomto

[3]Popis formátu PCRE sa dá dohľadať na adrese php.net/manual/en/reference.pcre.pattern.syntax.php

prípade to bude vetva s prázdnou postupnosťou, čiže na vstupe na nás nečaká viac znakov na spracovanie. V tomto bode sa končí vnorovanie a nasleduje proces vynorovania sa z rekurzie, pri ktorom dochádza k spracovaniu zvyšnej časti spracovávanej vetvy pravidla. A tak sa spracuje vetva bez rekurzie, po nej sa vynorí pravidlo o úroveň vyššie a spracuje vetvu, z ktorej sa rekurziou preklad dostal do bezrekurznej vetvy a po nej sa znova vynoríme vyššie, až sa dostaneme znova do úplne prvej volanej vetvy pravidla. Po jej spracovaní končí preklad. Rekurziu neskôr využijeme pri implementácii premenných v kapitole 4.2, nateraz nám ale stačí rozumieť princípu, na ktorom pracuje gramatika.

Pomocou týchto pravidiel v prvej etape YARA vykonáva funkcionalitu obyčajného kompilátora programovacích jazykov, ktorý pri prechádzaní YARA pravidla transformuje robustný vstupný zdrojový kód pravidiel na jednoduchý výstupný zdrojový kód pripomínajúci príkazy jazyka Assembly.

V skutočnosti ale ide o fiktívny jazyk, ktorého jediným cieľom je rozložiť jednotlivé pravidlá na postupnosť krokov skenu. Jeho význam spočíva v tom, že jedno pravidlo má obvykle dlhý životný cyklus, otestuje sa ním veľké kvantum súborov a preto je pre program neefektívne pri každom skenovaní čítať a analyzovať YARA pravidlá s ich zložitými konštrukciami. Výkonovo oveľa prívetivejšie je vygenerovať z pravidla jednoduchý medzikód, ktorý obsahuje na každom riadku práve 1 príkaz s argumentmi príkazu a následne pri opakovaní jednotlivých skenov čítať už len takto zjednodušené pravidlo. Vhodnou analógiou môže byť rozdiel medzi kompilovanými a interpretovanými jazykmi, kde v prípade kompilovaných súborov kompilátor odvedie prácu, ktorú pri spustení samotného programu už interpreter odvádzať nemusí a tým sa zrýchli jeho vykonávanie.

Toto čítanie už nepredstavuje výpočtovo náročnú oblasť výkonu programu a zlepší sa tým celkový čas vyhľadávania malvérov v kontrolovaných súboroch, o čo nám v konečnom výsledku ide.

2.4 Interpretácia medzikódu

Druhá etapa spočíva v interpretovaní vygenerovaného bajtkódu. Interpreter je implementovaný v súbore `exec.c` a je to konečný automat, ktorý v cykle prechádza jednotlivé príkazy a postupne nimi iteruje. Automat používa zásobník na uchovávanie dát v pamäti, a to nie len pre predávanie argumentov funkciám a ich volanie, ale aj vracanie návratových hodnôt z týchto funkcií.

Jednotlivé inštrukcie sú uložené v položke `code_start` obsiahnutej v objekte štruktúry `YR_RULES`, ktorý sme získali z prvej fáze behu. V nej sú sekvenčne uložené kódové označenia všetkých inštrukcií medzikódu. Hlava čítača sa nastaví na prvú položku `code_start`, nájde sa inštrukcia zodpovedajúca hodnote uloženej na danom zázname, vykoná sa telo inštrukcie a pri ďalšej iterácii sa posunie čítacia hlava na ďalšiu inštrukciu v poli inštrukcií. Tento cyklus sa opakuje až kým sa nevykonajú všetky inštrukcie uložené v sekvencii. Poslednou vykonanou inštrukciou bude príkaz `OP_HALT`, ktorý skontroluje prázdnosť zásobníka a následne ukončí interpretáciu medzikódu. Kontrola pre stav zásobníka je dôležitá pre odhalenie chýb v medzikóde, keďže sa využíva jeden zdieľaný zásobník pre všetky operácie v stavovom automate a teda jediný zabudnutý záznam na zásobníku uprostred kódu môže spôsobiť znefunkčnenie inštrukcií, ktoré po ňom ešte budú nasledovať a budú pracovať so zásobníkom. Táto kontrola teda slúži ako jedna z metód validácie očakávaného chovania kódu.

Príklad implementácie príkazu NOT: z vrchu zásobníka sa vyberie uložená hodnota, zneguje sa a vloží sa späť na vrch zásobníka.

```
switch(opcode) {
    ...
    case OP_NOT:
        pop(r1);

        if (is_undef(r1))
          r1.i = YR_UNDEFINED;
        else
          r1.i = !r1.i;

        push(r1);
        break;
    ...

}
```

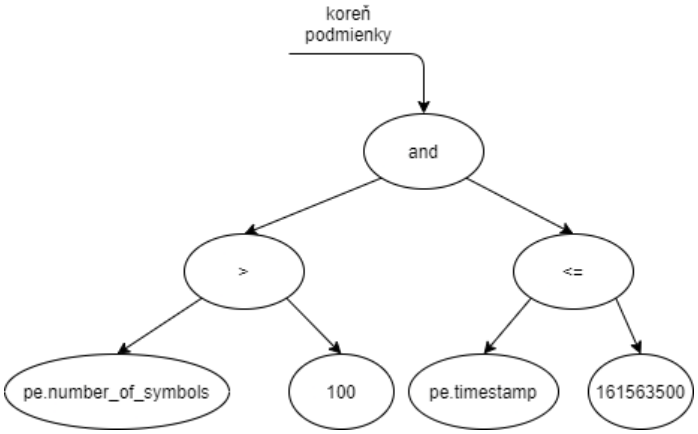
2.5 O nástrojoch využívajúcich YARU

YARA nie je len uzavretým nástrojom. V dnešnej dobe už slúži ako prostredie, ktoré v sebe integruje knižnicu `libyara`, ktorá vykonáva samotnú funkcionalitu programu. Vďaka tomu si každý môže naprogramovať svoju vlastnú YARU postavenú na rovnakej knižnici a prispôsobiť si nástroj tak, ako mu to vyhovuje.

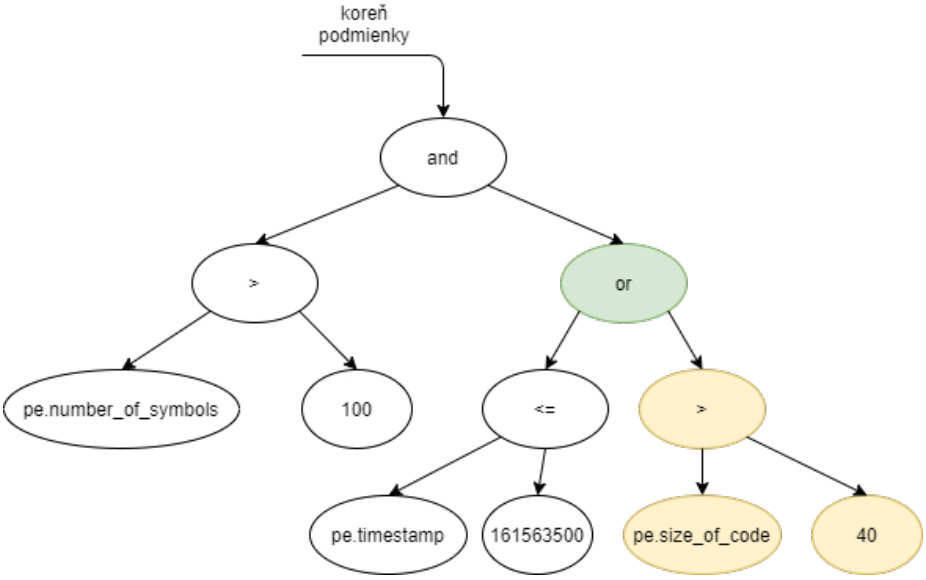
`Yaramod` je knižnica určená pre jazyky C++ a Python. Umožňuje nám načítať zo súboru YARA pravidlo, analyzovať ho a prevádzať nad ním operácie. Umožňuje nám napríklad nájsť v pravidlách istú syntaktickú konštrukciu a nahradiť ju za nejakú inú, čo je žiaduce pri transformácii pravidiel. Takéto transformácie sa robia napríklad v prípade, že chceme použiť YARA pravidlo na inej verzii YARY, ktorá nepodporuje konštrukcie použité v danom pravidle.

Yaramod prevádza textové pravidlo do abstraktného syntaktického stromu, v ktorom sa zachovávajú vzťahy medzi výrazmi a ich podvýrazmi, vďaka čomu sa dá do stromu vsunúť ďalšie elementy bez toho, aby sme museli manipulovať s celým pravidlom. To isté platí aj pre nahradenie elementov a ich mazanie. Tento modifikovaný strom je následne možné spätne previesť do textovej reprezentácie a exportovať ho do súboru.

```
// kód na transformáciu
pe.number_of_symbols > 100 and pe.timestamp <= 161583500
```



Obr. 2.2: Ukážka stromu v Yaramode.



Obr. 2.3: Ukážka modifikácie stromu.

```
// transformovaný kód
pe.number_of_symbols > 100 and
(pe.timestamp <= 161583500 or pe.size_of_code > 40)
```

Okrem modifikácie pravidla sa nad ním vďaka vygenerovanej stromovej štruktúre dajú robiť analýzy pravidiel a ďalej ich optimalizovať, prípadne s nimi pracovať cez iné externé nástroje, ktoré z rôznych dôvodov môžu potrebovať prístup k pravidlám na hlbšej úrovni, než len úrovni textu.

Yaramod obsahuje dva rôzne prístupy k práci s pravidlami. Tou prvou je načítanie existujúceho pravidla z textového súboru a následné prevádzanie operácií nad ním. K načítaniu takéhoto pravidla použijeme knihovňu Pog [4], ktorou definujeme pravidlá pre gramatiku

[4] https://github.com/metthal/pog

YARA pravidla. Prostredníctvom neho zadefinujeme novú syntax pre iterovateľné pole i premenné. Následne takéto pravidlo dokážeme načítať z reťazca.

```
yara_file = yaramod.Yaramod(yaramod.Features.Avast).parse_string('''
rule moje_pravidlo {
    meta:
        description = "This is a rule for testing"
    strings:
        $1 = "this is a string"
    condition:
        $1 at 50
}''')
```

Načítavať hotové pravidlá z reťazcov však nemusí byť vždy praktické, preto existuje aj varianta načítania cez `parse_file` s príslušným argumentom označujúcim názov súboru s pravidlami.

Okrem toho sa ale pravidlá dajú priamo vytvárať aj cez Yaramod prostredníctvom jeho komponenty `builder`. Príkladom pravidla vytvoreného skrz kód je nasledovný kód v jazyku Python využívajúci knižnicu Yaramod. Na ňom si rovno ukážeme aj to, akým spôsobom vytvoríme pravidlo z predchádzajúcej ukážky, tentoraz ale výlučne cez Python.

```
cond = (yaramod.match_at('$1', yaramod.int_val(50)).get()
rule = self.new_rule \
           .with_name('moje_pravidlo') \
           .with_string_meta('description', 'This is a rule for testing') \
           .with_plain_string('$1', 'this is a string.') \
           .with_condition(cond) \
           .get()
yara_file = self.new_file \
           .with_rule(rule) \
           .get()
```

Nad Yaramodom pracuje nástroj `Yara Rules Toolchain` (ďalej len YRTC), ktorý je interným nástrojom firmy Avast a jeho účelom je vykonávať operácie s pravidlami. Práve v nej sú definované úkony, ktoré sa bežne robia s internými malvérovými pravidlami. Význam takýchto operácií spočíva v tom, že i keď YARA spadá pod správu organizácie VirusTotal, súkromné firmy majú kvôli flexibilite vlastné verzie YARY, ktoré si sami vyvíjajú a pridávajú si vlastné vymoženosti, ktoré potrebujú. Z tohto dôvodu je pri pridávaní funkcionalít do firemnej YARY potrebné súčasne písať prekladač, ktorý pravidlo prevedie spätne do každým podporovanej formy zápisu. Okrem iného aj s touto funkcionalitou disponuje práve YRTC.

Kapitola 3

Návrh vylepšení

3.1 Iterovateľné podmienky

Malvéroví analytici pri svojej práci často potrebujú zapísať skupinu príznakov chovania malvéru, z ktorého skúmanému súboru postačuje splniť len niekoľko bodov pre to, aby sa súbor ako celok dal klasifikovať za daný malvér.

Problém spočíva v tom, že jazyk YARA pre toto neobsahuje žiadnu jednoduchú konštrukciu a preto používajú analytici v takýchto prípadoch cykly. Cyklus v sebe obsahuje optimalizačné prvky (o tých si povieme viac v kapitole 4), avšak už len samotná podstata iterovania cez cyklus pre kontrolu jednotlivých príznakov komplikuje zápis a čitateľnosť takéhoto pravidla.

```
rule moje_pravidlo {
  condition:
    for 1 i in (0..4): (
      (i == 0 and cuckoo.filesystem.file_access(/run\.sh/)) or
      (i == 1 and cuckoo.network.http_request(/http:\/\/web\.com/)) or
      (i == 2 and (pe.sections[0].raw_data_offset > 6)) or
      (i == 3 and cuckoo.network.http_request(/http:\/\/other\.com/))
    )
}
```

Z predchádzajúceho pravidla je zrejmé, že takýto zápis pravidla pôsobí neintuitívne, analytik si pri jeho písaní musí dávať pozor na číslovanie iterácií pri každom výraze a nesmie zabudnúť upraviť rozpätie, nad ktorým cyklus iteruje. Každá chyba môže spôsobiť nesprávne vyhodnocovanie pravidla a keďže obdobné pravidlá vôbec nie sú ojedinelé v obore malvérovej detekcie, musíme tento nedostatok YARY napraviť. Je žiaduce vytvoriť konštrukt, ktorý by dokázal vyjadriť funkčne zodpovedajúci výraz jednoduchšie čitateľnou formou.

```
rule moje_pravidlo {
  condition:
    1 of [ cuckoo.filesystem.file_access(/run\.sh/),
           cuckoo.network.http_request(/http:\/\/web\.com/),
           pe.sections[0].raw_data_offset > 6,
           cuckoo.network.http_request(/http:\/\/other\.com/) ]
}
```

Okrem numerického vyjadrenia počtu podvýrazov, ktoré musia byť splnené, preberieme z cyklov aj kľúčové slová `ANY` (ktorý je ekvivalentom čísla 1) a `ALL` (ktorý je ekvivalentom počtu podvýrazov[1]).

Syntax nenumerických kvantifikátorov a ich numerické alternatívy:

```
any of [vyraz1, vyraz2, vyraz3] = 1 of [vyraz1, vyraz2, vyraz3]
all of [vyraz1, vyraz2, vyraz3] = 3 of [vyraz1, vyraz2, vyraz3]
```

Význam nenumerických kvantifikátorov spočíva najmä v jednoduchej čitateľnosti pre ľudské oko, avšak v prípade kľúčového slova `ALL` nám zjednodušuje pridávanie a odoberanie podvýrazov bez toho, aby sme museli upravovať numerický kvantifikátor a mení sa aj ukončovacia podmienka, ale o tom si povieme viac až v implementácii v kapitole 4.

Prácu nám uľahčí fakt, že po funkčnej stránke pôjde stále o rovnaký princíp vyhodnocovania podvýrazov, čo nám umožňuje inšpirovať sa existujúcou implementáciou cyklov. Najväčšia zmena sa tak udeje na strane gramatiky jazyka.

3.2 Premenné

Pri písaní pravidiel často narazíme na problém, že potrebujeme opakovane pracovať s tým istým výsledkom operácie. Nech je to už výsledok matematickej operácie alebo návratová hodnota funkcie, obzvlášť pri zložitých pravidlách je nutné viacnásobne kopírovať tie isté riadky.

Toto predstavuje problém jednak vizuálny pre analytika, ktorému dlhé opakujúce sa časti kódu predlžujú a tým zneprehľadňujú pravidlá, ale je to zároveň aj problém výkonnostný, keďže sa operácie v kopírovaných riadkoch musia neustále opakovane vykonávať. Riešením by mohlo byť zavedenie premenných vrámci pravidiel.

[1] Ekvivalencia spočíva v chovaní príkazu, po implementačnej stránke sú rozdielne.

Príklad bez premenných:

```
rule moje_pravidlo {
    condition:
        math.entropy(
            pe.sections[0].raw_data_offset,
            pe.sections[0].raw_data_offset + pe.sections[0].raw_data_
                size) > 6 and
        math.entropy(
            pe.sections[0].raw_data_offset,
            pe.sections[0].raw_data_offset + pe.sections[0].raw_data_
                size) < 9
}
```

Príklad s premennými:

```
rule moje_pravidlo {
    variables:
        offset = pe.sections[0].raw_data_offset
        size = pe.sections[0].raw_data_size
        entropy = math.entropy(offset, offset + size)
    condition:
        entropy > 6 and entropy > 9
}
```

Výhodou tohto konceptu je to, že v momente spracovania definícií premenných sa vyhodnotia ich hodnoty, uložia sa do pamäte a pri ich použití v podmienkach sa už len pristupuje k výsledku v pamäti. Nedochádza tým k viacnásobnému vyhodnocovaniu.

V tejto časti práce budeme znova pracovať s gramatikou, oproti predchádzajúcej časti ale bude potrebné robiť ďalšie zmeny krížom celým kódom YARY, aby sa nám podarilo dosiahnuť náš cieľ. Narozdiel od operátora `of` ale nebudeme mať výhody možnosti inšpirovať sa iným funkčne takmer identickým konštruktom. YARA síce podporuje externé premenné, ktoré sú konštanty predávané YARE pri spustení programu cez prepínač, avšak tie sú prístupné zvnútra každého pravidla a majú charakter statických výrazov, teda výrazov, ktoré nie je potrebné vyhodnocovať za behu.

Koncept používania predom vyhodnoteného výrazu v podmienke sa istým spôsobom používa jedine napríklad pri hľadaní reťazcov v skenovaných súboroch, kde dojde k prehľadaniu súboru a následne sa kontroluje, či sa reťazec našiel v konkrétnej časti súboru. Aj v tomto prípade sa ale jedná o úplne iný prípad, keďže skenovanie prebieha ešte pred fázou behu medzikódu. Naše riešenie ale pracuje práve so znovupoužívaním výsledku, ktorý vznikne vykonávaním medzikódu.

Aj z posledných riadkov teda vyplýva, že stojíme pred úplne novým problémom, ktorý bude vyžadovať úplne nové riešenie.

3.3 Prehľadávanie behaviorálnych Cuckoo záznamov

3.3.1 Princíp hľadania reťazcov

Dlhodobým nasadzovaním YARY na skenovanie malvéru sa objavilo podozrenie, že skenovanie Cuckoo reportov spomaľuje vykonávanie jednotlivých skenov a na základe interného testovania bol určený za slabinu modulu samotný algoritmus, ktorým sa skenujú reporty

oproti bežným priamo analyzovaným súborom. Aby sme pochopili podstatu problému, popíšme si najprv proces skenovania súborov YAROU.

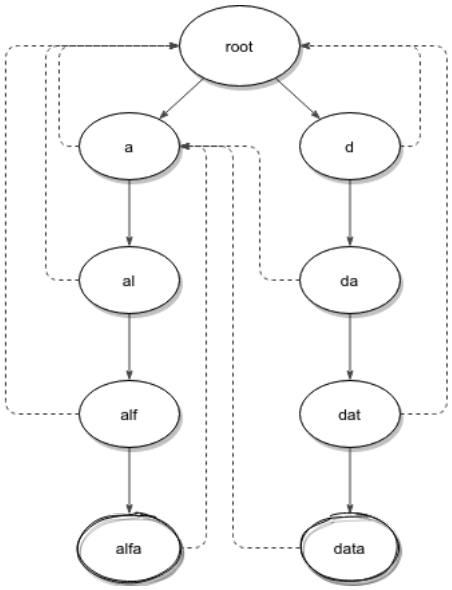
3.3.2 Prehľadávanie binárnych súboroch

Výrazy sa v nebehaviorálnych súboroch vyhľadávajú cez algoritmus `Aho-Corasick` [1] (ďalej len AC). Ide o rozšírenie algoritmu `Knuth-Morris-Pratt`, ktoré ho obohacuje o vyhľadávanie rôznych reťazcov naraz formou automatu.

V prvotnej fáze sa na základe vyhľadávaných slov vybuduje `Trie` pre všetky slová, ktoré budú zdieľať koreňový uzol. Následne sa vytvoria spätné hrany pre každý uzol (`failure links`), ktorými sa stavový automat vracia v prípade, že nenarazí na očakávané písmeno. V tomto prípade vedie spätná hrana na najdlhší možný sufix inej vetvy. V prípade, že neexistuje žiadna iná vetva, do ktorej uzla by sa AC mohol prepnúť, tak sa automat vracia do koreňového uzla a odznova začína s hľadaním výrazov. V prípade, že sa však na vstupne nachádza očakávaný znak, môžeme ďalej postupovať po doprovodných hranách smerom do koncoveho stavu, pri ktorom sa ohlási výskyt slova a zároveň sa ohlásia slová, ktore môžu byť prefixom práve nájdeného slova.

Nevýhoda algoritmu spočíva v réžii spôsobenej konštrukciou prechodov. Naopak, k jeho silným stránkam patrí rýchlosť prehľadávania, obzvlášť pri väčšom počte hľadaných slov v texte sa využíva jeho lineárna časová zložitosť `O(N + L + Z)`.

- N dĺžka textu
- L dĺžka hľadaných slov
- Z počet zhôd



Obr. 3.1: Ukážka konečného automatu Aho-Corasick pre vyhľadávanie slov `alfa` a `data`.

3.3.3 Prehľadávanie behaviorálnych súborov

Skenovanie reportov však nie je založené na princípe AC automatu. Namiesto toho sa tu využíva kombinácia knižnice Jansson [2] a Thompsonovho algoritmu na vyhľadávanie regulárnych výrazov v texte popísanom Rossom Coxom [2] (ďalej už len RE). Každý behaviorálny report je vo svojej podstate súbor vo formáte JSON, ktorý sa knižnicou Jansson preloží a načíta do programu.

Následne sa vďaka tejto knižnici pohodlne dostaneme napríklad k zoznamu mutexov, modifikovaným registrom alebo súborom, s ktorými skúmaná aplikácia manipulovala. V momente, kedy máme zúžený záber dát na typ hľadanej položky, začneme pomocou Thompsonovho algoritmu prehľadávať jednotlivé výrazy.

Pre každý hľadaný regulárny výraz sa vytvorí vlastné vlákno. Každé vlákno obsahuje ukazovateľ na začiatok vstupného reťazca a čítač inštrukcií, ktorý nám umožňuje postupovať cez sekvenciu inštrukcií v kóde.

Samotných inštrukcií je málo. Sú to:

- split
  - vytvorí dve vlákna, každé s čítačom inštrukcií nastaveným na inštrukciu z prvého, resp. druhého argumentu
- jmp
  - symbolizuje skok na inštrukciu z argumentu
- char
  - v argumente sa nachádza písmeno, ktoré sa musí zhodovať s písmenom uloženým na mieste, kam ukazuje reťazcový ukazovateľ; v tom prípade sa inkrementuje ukazovateľ na vstupný reťazec i čítač inštrukcií, inak sa ukončí vlákno
- match
  - nachádza sa na konci sekvencie pre regulárny výraz, hlási nájdenie výrazu v texte

[2] https://github.com/akheron/jansson

Výraz e		Kódová sekvencia
a		char a
e_1e_2		kód pre e_1
		kód pre e_2
$e_1\|e_2$		split L_1, L_2
	L_1	kód pre e_1
		jmp L3
	L_2	kód pre e_2
	L_3	
$e?$		split L_1, L_2
	L_1	kód pre e
	L_2	
$e*$	L_1	split L_2, L_3
	L_2	kód pre e
		jmp L_1
	L_3	
$e+$	L_1	kód pre e
		split L_1, L_3
	L_2	

Tabuľka 3.1: Sekvencie bajtového kódu pre regulárne výrazy. Prebraté z [2]

3.3.4 Ako zrýchliť prehľadávanie reportov

Jedným zo spôsobov, ako zrýchliť proces hľadania regulárnych výrazov v texte, je rozdeliť hľadanie na dve etapy. V tej prvej sa z regulárneho výrazu zoberie jeho časť, ktorá môže napríklad byť obyčajný reťazec. Tento reťazec sa vyhľadá v texte, čím spravíme tzv. `pre-filtering`, čiže zúžime záber dát, nad ktorými sa bude vykonávať druhá, spravidla komplikovanejšia a pomalšia etapa vyhľadávania regulárneho výrazu.

Zatiaľ čo zúženie záberu dát rýchlejšou a efektívnejšou metódou nám dodá rýchlosť, charakter dvojetapového prehľadávania nám ju uberá. Napríklad slovné reťazce sú predmetom vyhľadávania až dvakrát počas vyhľadávania. Prvý krát počas prvej etapy, kedy sa vyhľadáva výlučne táto časť výrazu a následne i počas druhej etapy, kedy je reťazec už súčasťou celého hľadaného výrazu. Toto predstavuje redundantnú prácu. Na druhej strane ale riešenie napriek tomu predstavuje zlepšenie na strane výkonu.

Príkladom knižnice, ktorá implementuje túto filozofiu, je Hyperscan od firmy Intel [4]. Tá poskytuje dva typy kompilovania regulárnych výrazov, prvá z nich (funkcia `hs_compile()`) skompiluje len jeden regulárny výraz do databázovej podoby potrebnej pre prehľadávanie obsahu. To nám nevyhovuje, keďže naším cieľom je prehľadať súbor na jeden prechod, aby sa minimalizovala réžia pri skenovaní. Prednosťou Hyperscanu je ale to, že ponúka aj možnosť kompilácie viacerých regulárnych výrazov do jednej a tej istej databázy (funkcia `hs_compile_multi()`). Tým sa dosiahne, že pri jednom prehľadávaní sa budú naraz hľadať všetky regulárne výrazy.

Už pri kompilovaní sa musíme rozhodnúť, v akom móde plánujeme skenovať vstupné dáta, keďže od toho závisí aj spôsob, akým sa budú hľadané výrazy ukladať do databázy. Dáta sa nám môžu všetky zmestiť do dočasnej vyrovnávacej pamäte, v ktorej bude prebiehať vyhľadávanie výrazov. Pri väčšom objeme dát sa ale môže stať, že sa nám do pamäte zmestí naraz len časť dát. Túto vyrovnávaciu pamäť, v ktorej máme časť dát, budeme nazývať

`blok`. Podľa toho, že v akej forme máme k dispozícii vstupné dáta, vyberieme vhodný mód, v ktorom bude pracovať skener:

- blokový
  - využíva sa, pokiaľ sú všetky prehľadávané dáta pripravené v jednom bloku
  - neukladá stav, umožňuje optimalizácie
  - pokiaľ už máme predalokované všetky dáta[3], blokový mód je najefektívnejší
- streaming
  - využíva sa, pokiaľ prehľadávané dáta presahujú jeden blok
  - tok dát sa rozdelí na bloky, každý blok ukladá svoj stav (`stream state`), aby sa dali identifikovať výrazy v presahoch medzi dvoma blokmi
  - bloky sa skenujú sekvenčne, nemusia byť všetky naraz dostupné
- vektorovaný
  - využíva sa, pokiaľ sú dáta na skenovanie dostupné vo viacerých nesekvenčne zoradených blokoch
  - neukladá sa stav

3.4 Yaramod a Yara Rules Toolchain

3.4.1 Yaramod

Rozšírenie Yaramodu spočíva v zabezpečení podpory pre nové prvky YARY. Prvým krokom bude rozšírenie gramatiky YARY pre načítanie pravidiel využívajúcich novopridané elementy `of` a definíciu premenných. Ponovom tak umožníme nástroju pracovať s pravidlami, ako je nasledovné:

```
yara_file = yaramod.Yaramod(yaramod.Features.Avast).parse_string('''
import "pe"
rule moje_pravidlo {
    strings:
        $1 = "Hello World"
    variables:
        subsys = pe.subsystem_version
    condition:
        1 of [subsys.major > 2 and $1 at 50, subsys.minor > 5 and $1 at 150]
}''')
```

Komponentu `builder` čakajú tiež zmeny. Pri syntaxi pre definíciu premenných sa inšpirujeme definíciami metadát, v prípade iterovateľných polí zase rozšírime výraz `of`, ktorý doteraz dokázal pracovať len s poliami reťazcov. Pravidlo z predchádzajúcej ukážky ponovom budeme môcť definovať nasledovne:

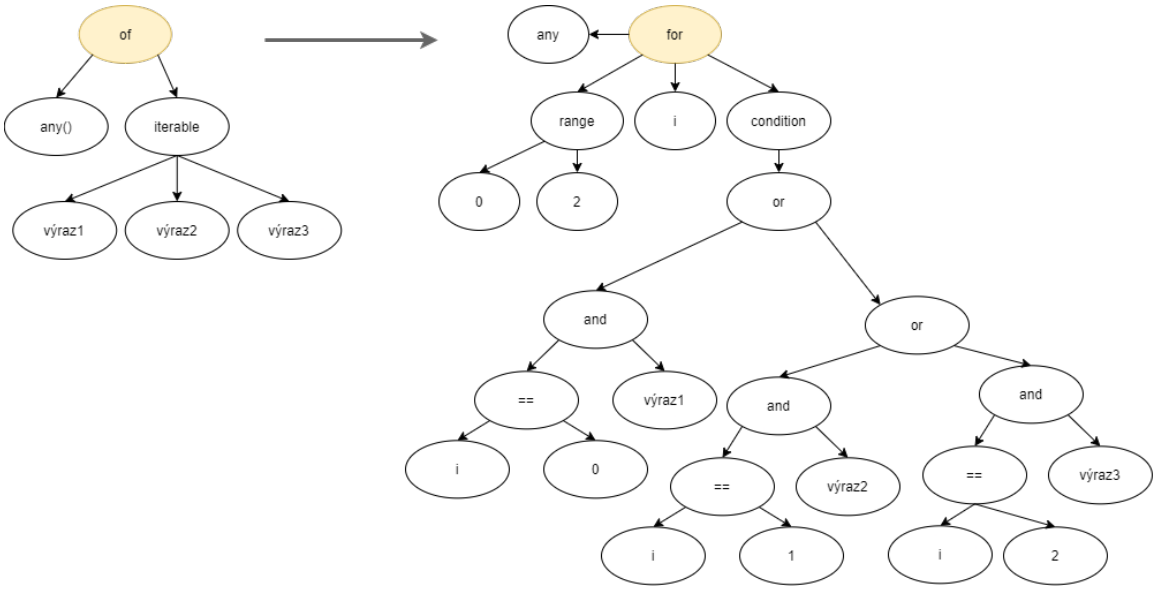
[3] Toto sa stáva napríklad v prípade, že si potrebujeme predom spracovať dáta.

```
cond = yaramod.of( \
    yaramod.int_val(1),\
      yaramod.iterable(\
        [ yaramod.id('subsys').access('major')() > yaramod.int_val(2) &&\
            yaramod.match_at(yaramod.id('$1'), yaramod.int_val(50)),\
          yaramod.id('subsys').access('major')() > yaramod.int_val(5) &&\
            yaramod.match_at(yaramod.id('$1'), yaramod.int_val(150)) ]))\
rule = self.new_rule \
           .with_name('moje_pravidlo') \
           .with_plain_string('$1', 'Hello World') \
           .with_struct_variable('subsys', 'time') \
           .with_condition(cond.get()) \
           .get()
yara_file = self.new_file \
           .with_module('pe') \
           .with_rule(rule) \
           .get()
```

Nové konštrukcie budú prístupné len v prípade, že je Yaramod spustený v móde pre našu verziu YARY- v tomto prípade YARY Avastu.

3.4.2 Yara Rules Toolchain

Od YRTC očakávame dve veci. Tou prvou je to, že prevedie `of` výraz na cyklus a nastaví mu správne parametre.



Obr. 3.2: Transformácia výrazu of na cyklus.

Druhou funkcionalitou je nájdenie premenných v podmienke, ich nahradenie za výraz uložený v premennej a zmazanie sekcie s definíciami premenných.

V prípade, že obe funkcie implementujeme správne, bude možné znova vygenerovať text pravidla, avšak nové pravidlo už bude používať výrazy podporované inými verziami YARY.

Kapitola 4

Implementácia

4.1 Operátor of

To, že sa v súčasnosti využíva na vyjadrenie výrazu s iterovateľným poľom pravdivostných výrazov práve cyklus, nie je náhoda. Disponuje totiž dôležitou vlastnosťou, ktorú požadujeme aj od iterovateľných premenných: efektívne vyhodnocovanie. Uvažujme o poli výrokov, ktoré obsahuje 5 výrokov, všetky pravdivé. Pokiaľ od tohto poľa očakávame, že bude obsahovať aspoň 3 pravdivé výroky, potom v záujme rýchlosti vykonávania programu musíme po nájdení 3. pravdivého výkonu prerušiť vyhodnocovanie ostatných výrokov, keďže podmienka je už v tomto bode splnená. Tento jav, v odbornej literatúre zvaný *skrátenené vyhodnocovanie (short-circuit evaluation)*, umožní skôr vyhodnotiť pravdivostnú hodnotu cyklu a skôr uvoľní prostriedky na vyhodnocovanie zvyšku pravidla.

4.1.1 Pamäťová štruktúra pre medzivýsledky

Pri cykloch ešte ostaneme, tie totiž pri vyhodnocovaní svojho tela ukladajú výsledok iterácie na haldu. Pre vyhodnotenie počtu pravdivých iterácií (iterácií tela cyklu, ktoré zanechajú na halde pravdivostný výsledok `TRUE`) si však musíme v pamäti uchovať i ďalšie údaje, napríklad momentálnu hodnotu iterátora, počet pravdivých iterácií alebo celkový počet iterácií (dôležité pre ukočenie iterovania). Keďže sa cykly môžu vzájomne vnorovať do seba, pre uschovanie týchto premenných potrebujeme robustné riešenie. Týmto riešením je vopred vyčlenené pamäťové pole, v ktorom sa pribúdajúcimi podcyklami (cyklami nachádzajúcimi sa v tele iných cyklov) bude posúvať index `var_frame` na čím ďalej, tým vyššie indexy. Pri ukončení výkonu podcyklu sa dekrementuje hodnota `var_frame`, ignorujú sa tým hodnoty uschované na tejto pozícii poľa a vrátime sa k stavu premenných z aktuálne najnišej úrovne- teraz už o jednu vyššej.

V našom prípade si budeme potrebovať uschovať nasledovné hodnoty:

- hodnotu pravdivých výrokov
- celkový počet výrokov
- minimálny požadovaný počet pravdivých výrokov ALEBO v prípade varianty príkazu s kľúčovým slovom `ALL` stačí pravdivostná hodnota posledného výrazu

...	var_frame + 0	var_frame + 1	var_frame + 2	...
	počet pravdivo vyhodnotených výrokov	počet vyhodnotených výrokov	[ANY] minimálny potrebný počet pravdivých výsledkov	
			[ALL] pravdivostná hodnota pre posledný vyhodnotený výrok	

Tabuľka 4.1: Využitie pamäťového priestoru pre varianty ANY a ALL

4.1.2 Algoritmus

Pri generovaní medzikódu musíme rozlišovať dve varianty príkazu `of`. Prvou je varianta `ANY`, pod ktorú spadajú príkazy, v ktorých sa vyskytuje `ANY` alebo číselný kvantifikátor. `ANY` je v tomto prípade aliasom kvantifikátora 1.

Príklad pseudokódu pre výraz `3 of [ expr1, expr2, expr3 ]`:

```
1       PUSH 3       ;
2       CLEAR_M 0    ; clear <expr> result accumulator
3       CLEAR_M 1    ; clear loop iteration counter
4       <expr1>      ; here goes the code for first array item expr,
                       its result will be at the  top of the stack
5       ADD_M 0      ; add boolean_expression result to accumulator
6       INCR_M 1     ; increment loop iteration counter
7       PUSH_M 2     ; primary expression minimum
8       PUSH_M 0     ; boolean_expression accumulator
9   .----JLE_P       ; jump to end if (minimum <= accumulator);
    |                  short circuit evaluation
    |    <expr2>     ; second array item code
    |    ...         ; same operations as for expr1
    | .--JLE_P
    | |  expr3 etc.  ; rest of array item expressions, operations
    | |                and jumps
10  `-+->SWAPUNDEF 1 ; if X is all, swap primary expression on stack for
                       number of iterations, otherwise don't do anything
11      PUSH_M 0     ; push the boolean_expression accumulator
12      INT_LE       ; compare boolean_expression accumulator to X
```

V prípade varianty výrazu `all of [ expr1, expr2, expr3 ]` sa mení ukončovacia podmienka vyhodnocovania podvýrazov:

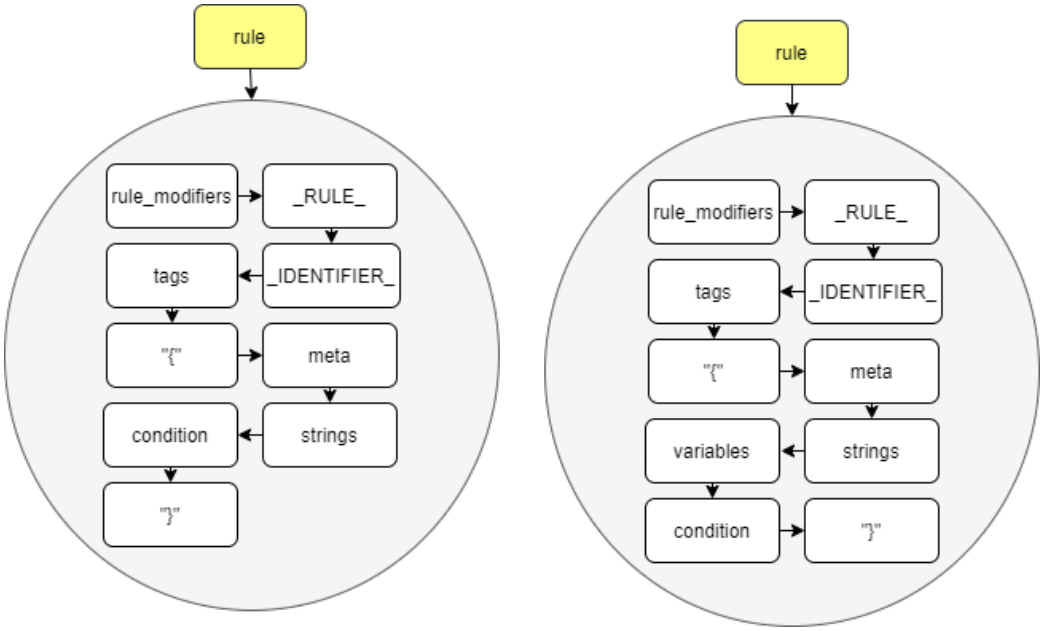
```
1       PUSH UNDEF   ; "all"
2       CLEAR_M 0    ; clear <expr> result accumulator
3       CLEAR_M 1    ; clear loop iteration counter
4       <expr1>      ; here goes the code for first array item expr,
                       its result will be at the  top of the stack
```

```
5       SET_M 3     ; save result of expr1 evaluation
6       ADD_M 0     ; add boolean_expression result to accumulator
7       INCR_M 1    ; increment loop iteration counter
8       PUSH_M 3    ; true/false result of exp1 resolution
9   .----JFALSE_P   ; jump to end if (minimum <= accumulator);
    |                 short circuit evaluation
    |    <expr2>    ; second array item code
    |    ...        ; same operations as for expr1
    | .--JFALSE_P
    | |  expr3 etc. ; rest of array item expressions, operations
    | |               and jumps
10  '-+->SWAPUNDEF 1 ; if X is all, swap primary expression on stack for
                       number of iterations, otherwise don't do anything
11      PUSH_M 0    ; push the boolean_expression accumulator
12      INT_LE      ; compare boolean_expression accumulator to X
```

4.2 Sekcia Variables

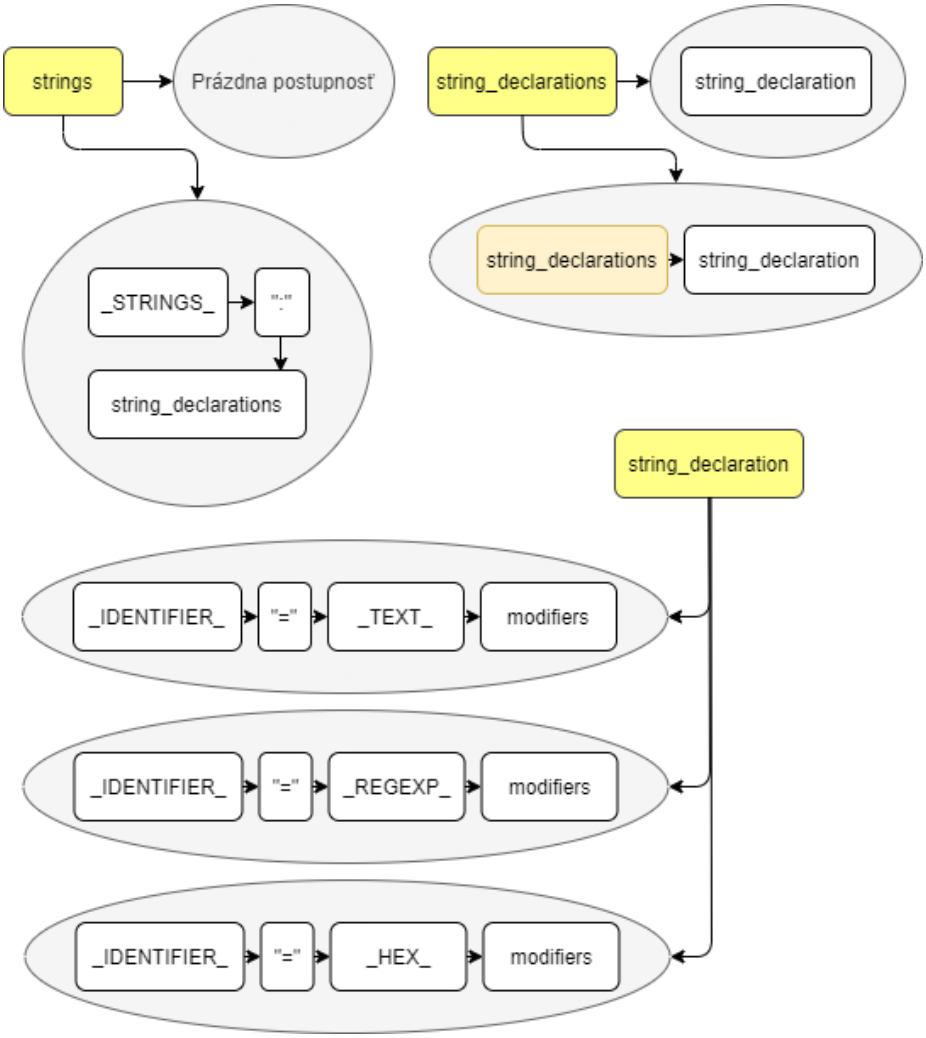
4.2.1 Gramatika

Ako sme už načrtli v kapitole 2.3, gramatika počíta s tým, že každý súbor sa skladá z niekoľkých pravidiel, pričom našou úlohou je implementovať lokálne premenné pre každé pravidlo zvlášť. Z tohto dôvodu si zaveďme nové pravidlo `variables`, ktoré zaradíme medzi definície reťazcov (položka `strings`) a podmienku `condition`.



Obr. 4.1: Porovnanie starej a novej gramatiky pravidla.

Pred implementáciou `variables` nahliadnime pre inšpiráciu do implementácie nepovinnej súčasti pravidla `strings`, keďže premenné budú predstavovať z pohľadu stavby pravidla podobnú položku.

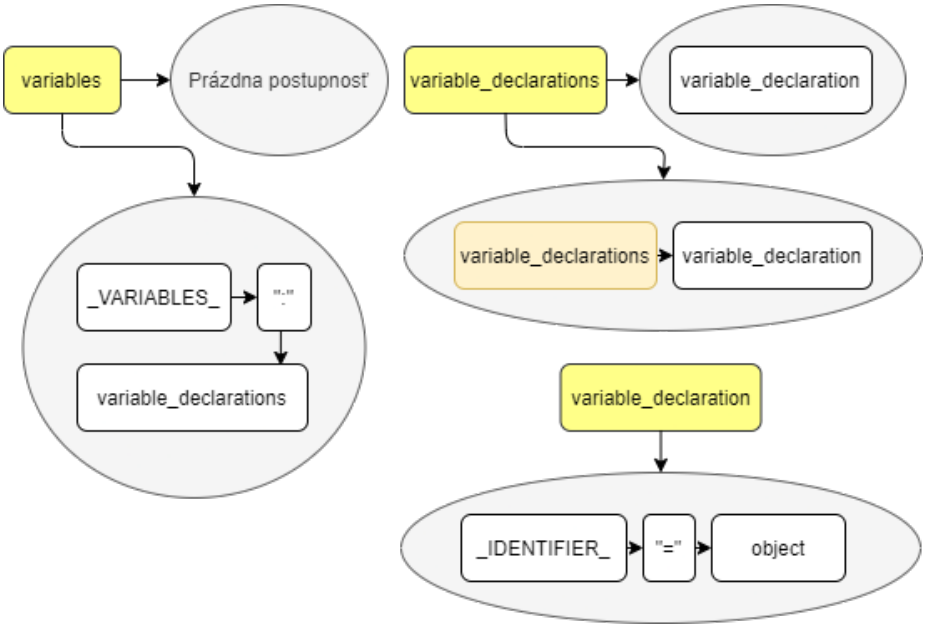


Obr. 4.2: Vizualizácia pravidla strings.

Môžeme si všimnúť, akým spôsobom je zabezpečená nepovinnosť reťazcovej sekcie v pravidle. Buď sa v definícii pravidla nachádza náveštie `"strings:"` a za ním neprázdna množina reťazcov, alebo sú reťazce z gramatiky vynechané. Vetva bez reťazcov je prázdna, čo naznačuje, že buď v pravidle nájdeme reťazcové náveštie, alebo tam očakávame až ďalšiu položku pravidla `condition`, ktorá sa volá z nadradeného pravidla `rule` po spracovaní pravidla `strings`, ktorá sa splnila vďaka prázdnej vetve.

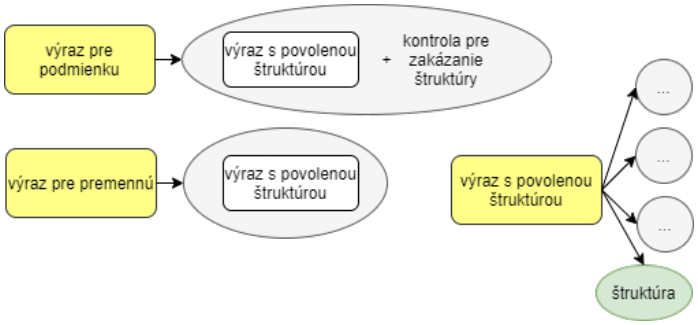
Ďalšou zaujímavosťou je spôsob, akým sa pravidlo postará o to, že v prípade nájdenia reťazcového náveštia musí nasledovať aspoň jedna definícia reťazca. Tu nachádza uplatnenie znova technika rekurzívy analyzovanej v kapitole 2.3, avšak tentoraz by sme tu darmo hľadali prázdne náveštie. to by nám totiž umožnilo nemať žiadne definície reťazcov po náveští. Namiesto prázdneho náveštia sa tu používa vetva s definíciou práve jedného reťazca. Vďaka tomuto triku sa pri viacnásobnej definícii rôznych reťazcov bude volať vetva s rekurzívou a pri poslednom reťazci sa zavolá vetva na spracovanie jednej deklarácie bez rekurzie. V prípade, že máme definovaný jediný reťazec, sa zavolá priamo vetva bez rekurzie. V prípade pravidla bez definície jediného reťazca sa pravidlo nebude môcť pohnúť do žiadnej vetvy a vráti chybu. Jediným spôsobom, ako nedefinovať reťazce, je teda úplne vynechať náveštie, keďže len v tomto pravidle existuje prázdna vetva.

Pri tvorení pravidla pre premenné sa teda môžeme inšpirovať reťazcami. Jediným spôsobom, ako nedefinovať žiadnu premennú, bude vynechať návešťie `"variables:"`, inak budeme od pravidla očakávať aspoň jednu definíciu.



Obr. 4.3: Vizualizácia pravidla variables.

Z pohľadu gramatiky nás čaká ale ešte jedna zmena. Narozdiel od zmien, ktoré sme robili doteraz, však bude oveľa invazívnejšia. Problém spočíva v tom, že keďže chceme ukladať ukladať do premenných výsledok výpočtu výrazu, ktorý sa doteraz musel viackrát počítať v podmienke, vo svojej podstate používame rovnakú gramatiku pre priradzované výrazy ako tie, ktoré používame v podmienke. Problém predstavujú štruktúry. Jednou z požiadaviek firmy na túto časť práce bola možnosť ukladať referencie na štruktúry do premenných. Štruktúry samé o sebe ale nemajú žiadnu návratovú hodnotu a teda sa ich nedá považovať za výraz v podmienke. Z tohto dôvodu sa v pravidle pre výraz vracia chyba v prípade, že použijeme identifikátor ukazujúci na neprimárny objekt[1]. Najjednoduchším riešením problému je povoliť štruktúry vo výrazoch a pri použití výrazu v pravidle spraviť dodatočnú kontrolu, či obsah podmienky nie je štruktúra.



Obr. 4.4: Utopické riešenie problému so štruktúrami.

[1]Primárne dátové typy sú reťazce, čísla, pravdivostné hodnoty. Do tejto kategórie nespadajú štruktúry, polia či slovníky.

Toto riešenie by ale obsahovalo závažnú chybu. Problém spočíva v tom, že kombináciou výrazov napríklad ich sčítaním vzniká ďalší výraz. Muselo by sa teda ošetriť pravidlo pre sčítanie. Tento istý problém ale vznikne pri veľkom počte iných operácií- odčítaní, násobení, delení, logickom AND a OR, argumentoch pre funkcie. A keď raz padne rozhodnutie podporovať okrem štruktúr aj polia, tak sa budú musieť znova robiť úpravy naprieč kódom. Správne riešenie preto bude zcela iné.

Je potrebné rozbiť pravidlo `expression` na viacero menších. Majme pravidlo `nonprimary_expression`, ktoré bude nástupcom originálneho pravidla. V ňom povolíme štruktúry. Nad týmto pravidlom sa postaví ďalšie pravidlo, ktoré nazvime `primary_expression`, v ktorom opäť prevedieme kontrolu pre štruktúru a vrátime pre ňu chybu. Ďalším pravidlom bude `complex_expression`, ktoré bude obsahovať všetky operácie s primárnymi výrazmi.

Následne vytvoríme posledné dve pravidlá. Prvým bude staronové `expression`, ktoré v jeho novej podobe bude kombinovať komplexný výraz a primárny výraz. Druhým pravidlom bude `object`, ktoré bude kombinovať komplexný výraz a neprimárny výraz, teda aj štruktúru. Pravidlo pre `expression` sa bude naďalej používať v podmienke, pravidlo pre `object` sa bude naopak používať pri definícii premenných. V prípade, že raz budeme potrebovať doplniť podporu pre ďalší neprimárny objekt v premennej, stačí ho povoliť v pravidle pre `nonprimary_expression` a zakázať ho v `primary_expression`.

4.2.2 Objekty a externé premenné

Pre pochopenie toho, na akom princípe budú pracovať premenné, musíme sa najprv pozrieť na to, ako funguje prístup k menným objektom v YARE.

Pre príklad si zoberme importované moduly. Hlavičkový súbor `compiler.h` obsahuje definíciu štruktúry `YR_COMPILER`. Objekt tejto štruktúry sa používa počas prekladu pravidla. Obsahuje rôzne dáta, pre nás ale bude dôležitá nasledovná časť:

```
typedef struct _YR_COMPILER
{
    ...
    YR_HASHTABLE* rules_table;
    YR_HASHTABLE* objects_table;
    YR_HASHTABLE* strings_table;
    ...
} YR_COMPILER;
```

Táto trojica tabuliek obsahuje odkazy na objektové reprezentácie pravidiel, objektov a reťazcov, na ktoré YARA narazí pri čítaní pravidla. V objektoch sa pritom nachádzajú všetky objekty pochádzajúce z importovaných modulov[2].

Pokiaľ pri preklade pravidla YARA narazí na identifikátor, začne hľadať jeho reprezentáciu v tejto hešovacej tabuľke. Pokiaľ existuje záznam s daným identifikátorom, vráti sa nájdený objekt. V opačnom prípade sa pokúsi nájsť pravidlo pod týmto menom a pokiaľ aj tento pokus zlyhá, preklad sa zastavuje a YARA oznamuje užívateľovi, že pravidlo obsahuje nedefinovaný identifikátor[3].

[2] Zoznam modulov je dostupný na adrese https://yara.readthedocs.io/en/stable/modules.html.

[3] Identifikátory reťazcov sú od iných identifikátorov odlíšené tým, že začínajú znakom $, preto sa pre ne v tomto prípade neprevádza kontrola.

YARA už istý typ premenných obsahuje. Jedná sa o externé premenné, ktoré sú definovateľné prostredníctvom prepínača pri spustení YARY. Pre príklad si uveďme definíciu celočíselnej premennej `myvar` s hodnotou 123:

```
./yara -d myvar=123
```

Záznamy externých premenných sú uložené v pamäti prostredníctvom `objects_table`. Problém nastáva v tom, že po ukončení prekladu prichádzame aj o celý objekt vrátane `YR_COMPILER`. Externé premenné potrebujeme ukladať aj do pamäte, ktorá sa prenesie do procesu skenovania. Z tohto dôvodu si definované premenné uložíme do záznamov `EXTERNAL_VARIABLE` a z nich si vytvoríme zoznam, ktorý si budeme ďalej uchovávať a na základe čoho bude možné rekonštruovať `objects_table` pre druhú fázu behu.

4.2.3 Interné premenné

V prípade nami definovaných premenných vo vnútri pravidiel (odteraz ich volajme interné premenné) nás však čaká podstatný rozdiel oproti tým externým. Interné premenné sú interné pre pravidlo, pod ktoré platia. Mimo neho neexistujú, a tak je možné vo viacerých pravidlách definovať rovnakú internú premennú s rovnakým identifikátorom, ale inou hodnotou a dátovým typom. Zatiaľ čo sa u objektu štruktúry `objects_table` očakáva, že sa bude zdieľať medzi pravidlami, zoznam lokálnych interných premenných sa bude vytvárať a mazať pri prechode z jedného pravidla do druhého.

Pre prehľadnosť je teda žiaduce neukladať interné pravidlá do `objects_table`, ale zavedieme si vlastnú hešovaciu tabuľku pre štruktúru `YR_COMPILER` s názvom `current_internal_variables_table`. Pri hľadaní identifikátora sa ponovom pozrieme najprv do tabuľky interných premenných a až následne do tabuliek objektov a pravidiel.

Toto nám umožní koexistenciu externých a interných premenných s rovnakým identifikátorom. V prípade, že dané pravidlo obsahuje internú premennú s daným menom, potom sa použije jeho definícia a prekryje sa externá premenná. V prípade, že interná premenná s daným menom v danom pravidle neexistuje, použije sa definícia externej premennej.

Majme súbor `pravidla.yar` obsahujúci nasledovné 2 pravidlá:

```
rule pravidlo1 {
    variables:
        moja_premenna = 1
    condition:
        moja_premenna > 0
}

rule pravidlo2 {
    condition:
        moja_premenna > 0
}
```

Pokiaľ toto pravidlo spustíme príkazom `./yara pravidla.yar pravidla.yar -d moja_premenna=0`, na výstup sa nám vypíše len `pravidlo1`.

Rovnako ako u externých premenných, aj pre interné premenné si zavedieme štruktúru `INTERNAL_VARIABLE`, jej objekty s jednotlivými premennými zreťazíme a uložíme do kontextu pre druhú fázu behu. V tej sa na začiatku znova vygenerujú objekty premenných pre všetky pravidlá. Všetky pravidlá budú mať znova vlastnú hešovaciu tabuľku.

Druhá fáza vykonáva bajtkód, ktorý nám vygenerovala prvá fáza pri preklade YARA pravidla. Vo svojej podstate je to konečný automat pracujúci na báze nekonečného cyklu a konštrukcie `switch`, kde každý príkaz má svoj `case`.

Pre nás bude mať špeciálny význam príkaz `OP_OBJ_LOAD`, ktorý načíta podľa identifikátora v argumente hodnotu premennej a zapíše ju na haldu. Tento príkaz upravíme znova tak, aby sa najprv pozrel do tabuľky interných premenných pre momentálne pravidlo a až potom hľadal identifikátor v tabuľke objektov.

Chýba nám však príkaz, ktorým môžeme priradiť hodnotu premennej. Nazvime ho `OP_OBJ_STORE`. Príkaz bude ukladať do identifikátora z argumentu príkazu hodnotu uloženú na halde. Týmto zabezpečíme, že sa do hešovacej tabuľky uloží vyhodnotená hodnota výrazu a k jeho výsledku už budeme môcť pristupovať cez túto tabuľku.

Pre jasnejšiu ilustráciu si popíšme `OP_OBJ_LOAD` a `OP_OBJ_STORE` pseudokódom:

```
switch aktualny_prikaz:
    ...
    case OP_OBJ_LOAD:
        objekt = najdi_internu_premennu(identifikator)

        if objekt sa nenasiel:
            objekt = najdi_objekt(identifikator)

        if objekt sa nenasiel:
            chyba

        push(objekt)
        break

    case OP_OBJ_STORE:
        objekt = najdi_internu_premennu(identifikator)
        if objekt sa nenasiel:
            chyba

        hodnota = pop()

        switch objekt->typ:
            case CELE_CISLO:
                zapis_cele_cislo_do_objektu(objekt, hodnota)
            case DESATINNE_CISLO:
                zapis_desatinne_cislo_do_objektu(objekt, hodnota)
            case RETAZEC:
                zapis_retazec_do_objektu(objekt, hodnota)
            case STRUKTURA:
                zapis_strukturu_do_objektu(objekt, hodnota)
            default:
                chyba
            break
        break
    ...
```

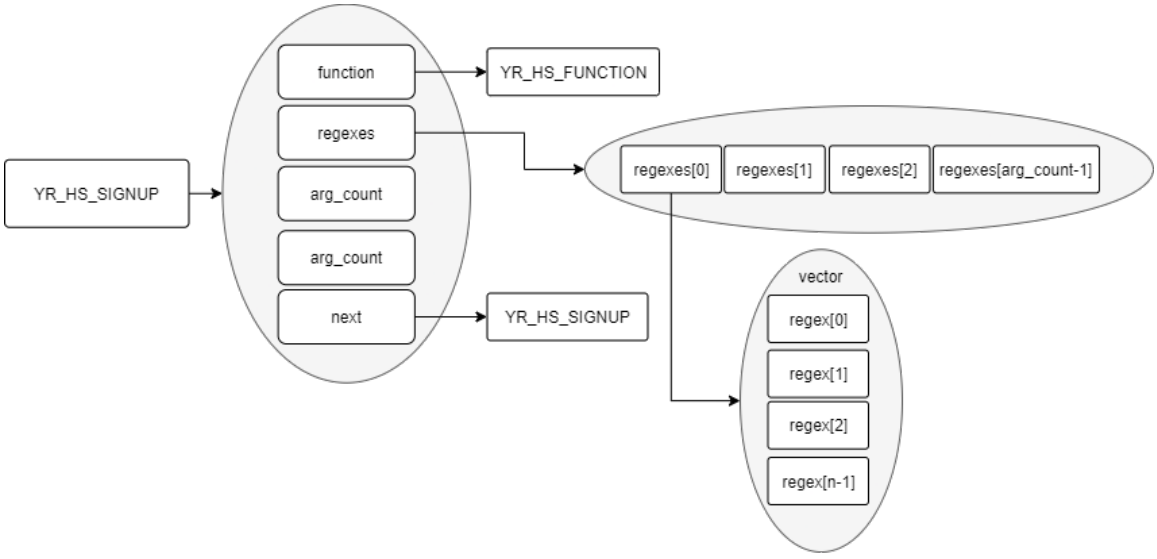
4.3 Hyperscan

Idea výhodnosti Hyperscanu je založená na tom, že sa vo fáze kompilácie pravidla vytvorí z jednotlivý hľadaných reťazcov skenovacia databáza (vo svojej podstate prehľadávací strom so spoločným koreňom) a následne sa prevedie sken každého dátového bloku reportu len raz. Prvá časť zahrňujúca budovanie databázy v čase kompilácie je časovo náročnejšia ako stávajúce riešenie, druhá časť prehľadávania v čase skenovania je naopak kratšia. Rozdiel vo výkonnosti skenu sa prehlbuje pri rastúcom množstve skenovaných dát v prospech Hyperscanu, obzvlášť pokiaľ dokážeme znovupoužiť jednotlivé databázy pre behy nad rôznymi dátami.

4.3.1 Konštrukcia databázy

Implementácia začína v gramatike (grammar.y). V nej sa v definícii `identifier` kontroluje vetva popisujúca volanie funkcie, čiže syntax `identifier(arguments)`. Po spracovaní identifikátora sa kontroluje, či voláme hyperscanovú funkciu. Tá sa pozná podľa toho, že pri deklarácii tejto funkcie v moduloch (v našom prípade sa jedná o modul `cuckoo`) sa namiesto bežnej deklaračného makra `declare_function` volá makro `declare_hs_function`. Rozdiel medzi týmito makrami spočíva v tom, že `declare_hs_function` okrem deklarácie objektu funkcie v pamäti zároveň označí túto funkciu za funkciu, ktorej argumenty typu regulárny výraz budú spracované Hyperscanom. Po identifikovaní volania HS funkcie sa do pomocnej pamäte kompilátora schovanej v objekte štruktúry `YR_COMPILER` ukladá pointer na objekt deklarovanej funkcie.

Spracovanie gramatiky pokračuje ďalej zátvorkami a argumentmi. Argumenty sú definované ako postupnosť `argument1, argument2, argument3, ...`, pričom jednotlivé argumenty z tejto postupnosti sú výrazy, z ktorých môže byť ľubovoľný argument regulárny výraz. Regulárny výraz je definovaný pravidlom `regexp`. Túto definíciu upravíme tak, aby v prípade, že máme v pamäti z predchádzajúceho kroku odkaz na funkciu - a teda spracovávavame hyperscanovú funkciu, sme načítali hodnotu tohto argumentu a vytvorili v pamäti objekt štruktúry `YR_HS_SIGNUP`, ktorá obsahuje uložené reťazce pre volania tej istej funkcie.



Obr. 4.5: Štruktúra SIGNUP.

Po spracovaní celého pravidla sa ešte pred ukončením kompilátora YARY transformujú jednotlivé objekty `YR_HS_SIGNUP` do objektov štruktúry `YR_HS_FUNCTION`, ktorá bude obsahovať hyperscanovú databázu pre každý argument funkcie. Tieto databázy prežijú beh kompilátora a posunú sa do behu skenu vrámci objektu štruktúry `YR_RULES`.

4.3.2 Efektívnejšia alokácia pamäte a vektory

Pri zhromaždovaní argumentov pre tie isté funkcie sme narazili na problém. Vopred totiž nevieme, koľko volaní tej istej funkcie vlastne očakávame. Z tohto dôvodu sa nedá dopredu určiť, o akej veľkosti bude pole reťazcov, ktoré zhromažďujeme a ktoré konzumuje funkcia `hs_compile_database` slúžiaca na vytvorenie databázy. Ponúka sa možnosť vždy použiť `realloc` na existujúce pole v prípade, že ho chceme rozšíriť, avšak proces realokácie pamäte je náročnou operáciou a najmä na veľkej škále skenovania predstavuje problém, ktorý môže zapríčiniť pomalšie vykonávanie programu, než by sme to očakávali. Potrebujeme teda znížiť počet manipulácií s pamäťou.

Odpoveďou na tento problém je vytvorenie vyššej softvérovej vrstvy pre alokáciu pamäte. Spravíme si vlastnú implementáciu funkcie `malloc`, tzv. `sub-alokátor`, ktorý bude fungovať veľmi podobne ako skutočný alokátor. Rozdiel bude ale v tom, že pokiaľ budeme volať náš vlastný softvérový malloc, potom sa skutočný malloc zavolá menej krát, keďže náš malloc si bude sám riešiť prerozdeľovanie už alokovaného priestoru a vytváranie pamäťovej rezervy. Inými slovami si naša implementácia mallocu predalokuje väčší blok pamäte, ktorý následne sám rozdeľuje. Nezmenší sa tým síce pamäťová náročnosť programu (v skutočnosti sa tým navýši, ale k tomu sa dostaneme neskôr), ale pokiaľ sa už program rozhodne zavolať alokátor, potom k samotnej nízkoúrovňovej alokácii dojde menej ráz. Podobný princíp využíva aj alokátor z knižnice `boost` pre jazyk C++. Výhodou tohto prístupu je to, že si sami určíme stratégiu alokácie pamäte pri rastúcom objeme dát na uloženie a túto funkciu neprenechávame operačnému systému, ktorý si pamäť predalokuje väčšinou na báze násobkov veľkosti stránky disku.

Pre lepšiu predstavu si predstavme modelovú situáciu a rozdielne chovanie jednovrstvovej alokácie a dvojvrstvovej alokácie na praktickej ukážke.

Obr. 4.6: Vizualizácia obyčajného jednovrstvového alokátora.

Jednovrstvové riešenie pracuje priamo so systémovým alokátorom. Predpokladajme, že máme alokovaný blok pamäte pre 4 položky. Túto pamäť chceme rozšíriť pre piatu položku, a tak zavoláme `realloc` pre existujúci blok pamäte a predáme mu novú požadovanú veľkosť bloku. Táto veľkosť sa následne zaokrúhli nahor tak, aby zodpovedala násobkom veľkosti stránok (konkrétna hodnota veľkosti ale závisí od implementácie). Nový blok pamäte sa rozšíri a v závislosti od zvyšku adresného priestoru buď ostane na rovnakom mieste v pamäti,

alebo sa blok premiestni do inej časti pamäte, kde bude preň dosť miesta. V prípade, že budeme takto neustále pridávať po jednej položke do poľa, narazíme na problém so spomalením spôsobeným realokáciou a presunom dát po úložisku, ktorý sme si už vyššie popísali. Tento problém nám pomôže vyriešiť medzivrstva, vďaka ktorej sa nemusí dojsť k toľkým realokáciám.

Obr. 4.7: Vizualizácia dvojvrstvového alokátora.

V tej sa predalokuje vždy väčší blok pamäte a tento priestor sa následne cez túto medzivrstvu prerozdeľuje. Testovaním sa zistilo, že najčastejšie budeme vytvárať polia o veľkosti niekoľko desiatok záznamov, ale pri niektorých funkciách sa počty argumentov navýšia na niekoľko tisícok záznamov. Práve pri týchto obrovských poliach sa naplno prejaví výhoda predalokácie a znížené množstvo presunov dát po úložisku.

Logiku dvoch stupňov máme vymyslenú, avšak stále nám chýba mechanizmus, ktorý túto pamäť bude prerozdeľovať. Preto si vytvoríme štruktúru `YR_VECTOR`. Tá bude obsahovať obsadenú pamäť a celkové alokované množstvo pamäte na prerozdelenie. Pri alokácii pamäte sa alokuje miesto pre `YR_VECTOR` a následne aj pre prerozdelovanú pamäť. Následne k nemu dopíšeme funkcie `yr_vector_create`, `yr_vector_push` a `yr_vector_destroy`, ktoré ako už z názvu vyplýva, riadia proces vytvorenia vektoru (prvotná alokácia priestoru), jeho plnenia dátami (inkrementovania jeho kapacity a prípadnej realokácie priestoru) a zmazanie vektoru (uvoľnenie všetkej pamäte). Stratégia, ktorú použijeme, počíta s dvojnásobným rastom kapacity vektoru pri každej realokácii. Z tohto vyplýva, že pri skutočne vysokom objeme dát sa budú realokácie diať čím ďalej, tým menej a rozdiel oproti použitiu obyčajnej realokácie bude viac a viac markantný.

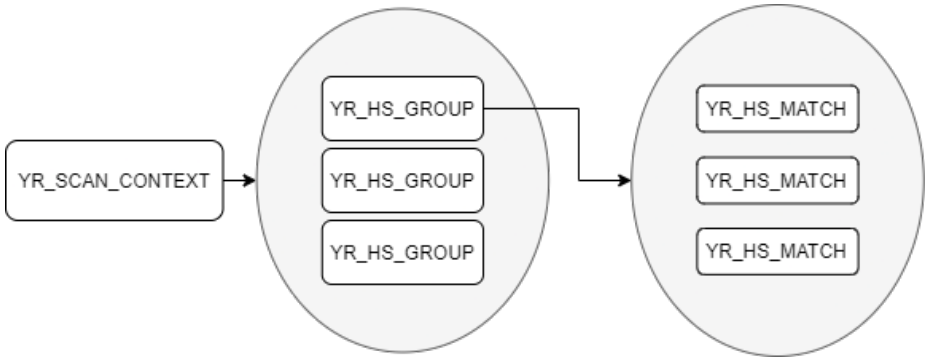
Veľmi podobný princíp využíva `YR_NOTEBOOK`, avšak ten namiesto realokácií existujúceho bloku alokuje v prípade, že dojde rezervovaná pamäť, ďalšiu stránku pamäte identickej veľkosti. Výhoda tohto prístupu spočíva v tom, že pri veľkom objeme dát vznikne namiesto súvislého kolosu alokovaného priestoru pamäte viacero malých blokov dát, pre ktoré sa jednoduchšie nájde voľná pamäť. V našom prípade ale potrebujeme práve jeden súvislý blok dát, keďže `hs_compile_databases` vyžaduje pole, ktoré sa na disku ukladá sekvenčne.

4.3.3 Skenovanie dát

Druhá fáza behu YARY môže, ale nemusí prísť okamžite po prvej. Z tohto dôvodu sa musia nanovo deklarovať jednotlivé funkcie modulov. Keďže tie sa budú deklarovať znova v rovnakom poradí, ako tomu bolo v prvej fáze, tak je zaručené, že sa im podľa rovnakého algoritmu priradí znova ten istý hyperscanový identifikátor funkcie. Pokiaľ sa počas vykonávania bajtkódu narazí na volanie funkcie, ktorej hyperscanový identifikátor zodpovedá

niektorému objektu `YR_HS_FUNCTION`, potom sa zahajuje proces skenovania technológiou Hyperscan.

Prehľadá sa každý relevantný záznam v poskytnutom Cuckoo reporte nad databázami prislúchajúcimi danej funkcii. Tieto databázy zahrňujú všetky volania funkcie so všetkými argumentmi poskytnutými tejto funkcii. V pamäti (tentoraz objekt štruktúry `YR_SCANNER`) sa vytvorí záznam `YR_HS_GROUP`, do ktorého sa budú zapisovať všetky nájdené zhody. Po dobehnutí tohto skenu skontrolujeme, či momentálne spracovávané volanie funkcie (ktoré má svoj unikátny identifikátor vrámci danej funkcie) zachytilo nájdené zhody v reporte. Pokiaľ áno, potom návratová hodnota volania funkcie je `true`, inak je `false`. Pri dalších volaniach tejto istej funkcie skenovacia funkcia objaví existenciu záznamu pre túto funkciu. Podľa toho YARA pozná, že sken pre túto funkciu už prebehol a môžeme prejsť priamo na hľadanie zhody pre aktuálne volanie funkcie. Pokiaľ sa zhoda nájde, vráti sa `true`, inak opäť `false`.



Obr. 4.8: Vizualizácia pravidla variables.

Po dobehnutí druhej a finálnej fázy prichádza čas na zničenie skenera- objektu štruktúry `YR_SCANNER`. V tejto fáze sa zahadzujú všetky zhody, záznamy pre funkcie, jednotlivé objekty `YR_HS_FUNCTION`, ale aj samotné databázy, ktoré na konci dňa zaberajú najviac miesta v pamäti.

4.4 Yaramod a YRTC

V implementácii nových konštruktov sa musíme na zmeny pozerať vo dvoch rovinách. Tou prvou je načítanie hotového pravidla cez gramatiku postavenú na knižnici `Pog`. Naším cieľom je teda upraviť pravidlo `rule`, ktoré sa rozdvojí. Jedna jeho vetva pre stávajúcu verziu YARY ostane nedotknutá. V jeho druhej vetve určenej pre beh v móde pre našu verziu YARY sa ale medzi voliteľné náveštia `metas` a `strings` pridá `variables`. V tele definície pravidla pre `variables` sa následne priradí identifikátoru `symbol`.

4.4.1 Symboly

Symbol je nositeľom významu prvku. Umožňuje nám rozlišovať volanie funkcie od poľa, slovníka od štruktúry. Dodáva kontext práve načítavanej časti pravidla, vďaka čomu dokážeme zistiť, akého typu je atribút konkrétnej štruktúry a či vôbec existuje atribút s týmto názvom.

V prípade bežných premenných sa im bude priradzovať symbol typu `ValueSymbol`, ktorý bude obsahovať originálny výraz aj s jeho typom. V prípade štruktúr ale narazíme

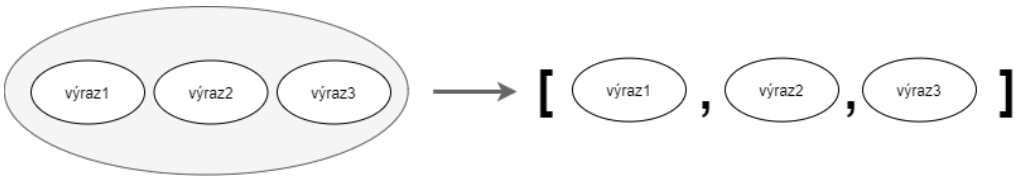
na problém, keďže tie nemajú návratovú hodnotu, naopak v ich prípade potrebujeme zachovať informáciu, k akým atribútom štruktúry vieme pristupovať a aké vlastnosti majú tieto prvky.

Táto štruktúra je už raz v pamäti uložená pod svojim menom, na základe ktorého k nemu vieme pristúpiť. Pokiaľ chceme vedieť k tomuto objektu pristúpiť aj pod alternatívnym menom - napríklad identifikátorom našej premennej, môžeme tento objekt buď celý skopírovať a znova vložiť pod iným kľúčom do pamäte, alebo vytvoriť referenčný symbol s kľúčom zodpovedajúcim identifikátoru premennej, ktorý bude vykonávať funkciu ukazovateľa na hlavnú štruktúru s pôvodným názvom.

Práve druhá metóda je tá, ktorá predstavuje transparentnejšiu implementáciu, a preto vytvárame nový typ symbolu `ReferenceSymbol`. A tak v prípade, že pri načítaní podmienky narazíme na identifikátor premennej, Yaramod nájde pod kľúčom premennej objekt referencie, cez ktorú sa dostane do originálneho symbolu `StructureSymbol` a ďalej pracuje so symbolom tak, ako by sme v podmienke používali priamo túto štruktúru.

4.4.2 Výrazy

Výrazy sú nositeľmi dátového typu a hodnoty. V prípade `OfExpression` výraz zastrešuje pole iných výrazov, cez ktoré sa pravidlo iteruje. Týchto výrazov sa priamo ani nemusíme dotýkať, keďže naša snaha spracovať pravidlo zlyhávama nie na kľúčovom slove `of`, ale na tom, že YARA neobsahuje výraz, ktorý zastrešuje pole podvýrazov ohraničených hranatými zátvorkami. A tak musí vzniknúť výraz `IterableExpression`. Jeho jedinou úlohou bude to, aby dokázal obsiahnuť podvýrazy, ktoré sú v ňom uložené, oddelil ich čiarkou a zároveň zachoval ohraničenie hranatými zátvorkami, až sa pravidlo rozhodneme znova previesť späť na text.



Obr. 4.9: Úloha výrazu Iterable.

4.4.3 Builder

Druhou rovinou zmien v Yaramode je rovina tvorby pravidiel cez kód. Symboly a výrazy, ktoré sme pridali a napojili na gramatiku Pogu, musíme sprístupniť aj programovacím jazykom. Pre dosiahnutie tohto cieľa zadefinujeme v súbore `yara_rule_builder.cpp` funkciu `iterable`, ktorá bude vracať hodnotu typu `YaraExpressionBuilder`. Argumentom tejto funkcie bude pole podvýrazov. Táto funkcia následne vygeneruje výraz, ktorý bude obsahovať všetky prvky tohto poľa oddelené čiarkou a ohraničené hranatými zátvorkami, čiže to isté, čo sme robili v gramatike pre pravidlo vo forme reťazca.

Následne potrebujeme zabezpečiť aj definíciu nových premenných. K tomu si vytvoríme ďalšie funkcie:

- withStringVariable(const std::string key, const std::string value)
- withIntVariable(const std::string key, std::int64_t value)

- withUIntVariable(const std::string key, std::uint64_t value)
- withHexIntVariable(const std::string key, std::uint64_t value)
- withDoubleVariable(const std::string key, double value)
- withBoolVariable(const std::string key, bool value)
- withStructVariable(const std::string key, const std::string identifier)

Tie nám vložia do pravidla výraz `value` pod identifikátorom `key`. V prípade štruktúr označuje argument `identifier` identifikátor štruktúry, na ktorú potrebujeme vytvoriť referenciu.

Finálnym krokom je sprístupniť tieto funkcie aj knižnici pre Python, preto v súbore `yaramod_python.cpp` spárujeme syntaktické pravidlá s novými funkciami a dátovými typmi, ktoré sme pridali. Vďaka tomu umožníme klientom našej knižnice pracovať so zoznamom premenných pre pravidlo, pridávať ďalšie nové premenné, ale zároveň ich aj všetky zmazať. Umožníme im definovať výrazy využívajúce `ReferenceSymbol`, `IterableExpression` a jeho podvýrazy.

4.4.4 Prevod pravidiel na kompatibilný formát

Implementácia prekladu syntaktických konštrukcií v YRTC už len pracuje s abstraktným stromom prostredníctvom zavolania príkazu `modify` nad podmienkou. Tým sa spustí prehľadávanie stromu zvrchu a začne sa zanorovanie do podvýrazov. Pre každé pravidlo nahradíme výraz tohto typu za univerzálny cyklus prostredníctvom `ModifyingVisitor`, čím vymeníme jeden list stromu za iný, následne prevedieme modifikáciu postupnosti lexém `TokenStream` tak, aby reflektoval novú hierarchiu a nakoniec znova prevedieme strom pomocou postupnosti lexém na text.

To isté spravíme aj v prípade premenných, najprv si uložíme všetky definície premenných, následne v podmienkach vyhľadáme identifikátory, porovnáme všetky identifikátory `IdExpression` voči nášmu zoznamu premenných a v prípade zhody nahradíme tento identifikátor za pôvodný výraz. Nakoniec po nahradení všetkých referencií za premenné zmažeme aj samotné definície premenných, čím efektívne vrátime pravidlo späť do zápisu, ktorý analytici bežne používali pred touto bakalárskou prácou.

Kapitola 5

Vyhodnotenie

5.1 Operátor of

V dnešnej dobe analytikom vo firme Avast odporúča využívať tento konštrukt namiesto cyklov. Dôvodov je viacero, tým najjednoduchšie merateľným je dĺžka pravidla. Naše pravidlo z kapitoly 3.1 novozavedený konštrukt skrátil na `75%` pôvodného počtu znakov. Pokiaľ zoberieme do úvahy len dĺžku podmienky v starom a novom pravidle, potom sa v novej podmienke dostávame na `72%` dĺžky pôvodnej podmienky.

Ďalšími nemerateľnými benefitmi sú:

- lepšia čitateľnosť pravidla
- jednoduchšie pridanie ďalšieho výrazu
- odstránenie potreby indexovať výrazy pre iterovanie výrazmi
- odstránenie potenciálneho zdroja chýb známych z cyklov, ktoré plynú z toho, že analytik zabudne prepísať maximálnu hodnotu indexu, ktorú cyklus môže nadobudnúť počas iterovania, prípadne ho nastaví na nesprávnu hodnotu a niektoré výrazy sa kvôli tomu nikdy nevykonajú

Funkcia nového operátora je kozmetická, jeho úlohou je umožnenie alternatívneho zápisu konštruktu, ktorý sa už aj pred tým dal zapísať inou syntaxou. Z tohto dôvodu nie je možné prezentovať žiadne výkonové rozdiely oproti stávajúcemu riešeniu.

5.2 Premenné

Hlavným prínosom interných premenných je ľahšia čitateľnosť. Ich benefit sme si predstavili už v ukážkach v kapitole 3.2. Ich ďalšou výhodou je ale šetrenie výkonom, keďže sa predchádza znovuvyhodnocovaniu výrazov, ktoré už raz vyhodnotené boli. Práve tento pozitívny dopad je i merateľný, avšak rozdiely sú v skutočnom prostredí tak nízke, že kvôli zaokrúhľovaniu nám ich naša meracia metóda nedokáže zachytiť a dokázať zlepšenie môžeme jedine ak si pomôžeme prispôsobením si pravidla.

Zoberme si rovnakú podmienku, ako v kapitole 3.2, ale zreťazme ich za sebou cez operátor `OR` stokrát. To nám dostatočne predĺži vykonávanie kódu, aby sme dokázateľne dosiahli časové zlepšenie.

Bez premenných [ms]	S premennými [ms]	Relatívny podiel [%]
0.251	0.180	71.71
0.224	0.161	71.88
0.254	0.188	74.02
0.223	0.161	72.20
0.226	0.155	68.58
0.222	0.159	71.62

Tabuľka 5.1: Porovnanie skenovania pravidlom nad súborom.

Z tabuľky 5.1 vyplýva, že sa za umelo vytvorených podmienok (pravidiel) dá prezentovať výrazná úspora výkonu pri vykonávaní skenovania. Pri skutočnom skenovaní bude ale tento rozdiel menej patrný, keďže sa ten istý výraz bude v tom istom pravidle opakovať veľmi málo krát. Aj vďaka tomu budú rozdiely nemerateľné aj napriek tomu, že sme práve dokázali zrýchlenie na veľkej škále.

5.3 Hyperscan

Pri testovaní vplyvu novej skenovacej metódy na rýchlosť sa použije knižnica `time.h` obsahujúca metódu `clock()`. Pomocou tejto metódy odmeriame časový rámec, v ktorom sa vykonáva bajtkód v druhej fáze skenu rozdielom hodnoty `clock()` na konci výkonu bajtkódu a na jeho začiatku. Túto hodnotu následne ešte predelíme hodnotou `CLOCKS_PER_SEC`, vďaka ktorej dostaneme časový odhad výkonu kódu v sekundách.

Túto hodnotu následne vypíšeme do konzoly a výstup necháme skontrolovať našim skriptom, ktorý sčíta časové intervaly pre jednotlivé behy YARY nad súbormi, aby sme získali celkovú dĺžku behu programu. Celkovú dĺžku behu neporovnávame z dôvodu, že u Hyperscanu vo fáze kompilácie prebieha budovanie databáz, ktoré je časovo náročné a ktoré by nám pokrivilo naše výsledky. Keďže je naším cieľom zefektívniť len druhú fázu (čiže tú, ktorá sa bude neustále opakovať), tak nám na dĺžke behu prvej fázy nezáleži a teda nie je objektom nášho záujmu.

V prostredí Avastu predstavuje výkonovo najnáročnejšiu časť opakovaný beh druhej fázy skenu nad rôznymi reportmi, z tohto dôvodu budeme toto prostredie emulovať použitím nie jedného gigantického súboru na skenovanie, ale veľkého množstva malých reportov. Z produkčných serverov firmy preto najprv použijeme vzorku o veľkosti 100 reportov. Tie spustíme nad naším testovacím pravidlom, ktoré bolo umelo zložené nad tisíckami rôznych výrazov,ktoré sa pri bežných skenoch využívajú.

Dôvod, pre ktorý nemeriame výlučne vykonávanie funkcie skenujúcej report bez spracovania zvyšku bajtkódu, je ten, že pri tomto štýle testovania sa vo výsledkoch objavovalo množstvo malých čísel, ktoré sa zaokrúhľovali, čo malo za následok, že po ich sčítaní by sme dostali väčší súčet, než pri meraní celej druhej fázy. Takýto údaj samozrejme nedáva zmysel a i keď opticky vylepšoval efektivitu nášho programu, zahrnutie celej druhej fázy do výpočtov nám napriek tomu poskytuje výsledky bližšie k realite, aj keď to znamená to, že neporovnávame len rýchlosť skenovania reportov, ale aj celé fázy skenovania (čiže aj spracovania inštrukcií nesúvisiacich priamo so skenovaním reportov).

```
import "cuckoo"

rule unknown
{
    condition:
        cuckoo.sync.mutex(/vyraz1/) or
        cuckoo.sync.mutex(/vyraz2/) or
        ...
        cuckoo.sync.event(/vyraz3/) or
        cuckoo.sync.event(/vyraz4/) or
        ...
        cuckoo.sync.semaphore(/vyraz5/) or
        cuckoo.sync.semaphore(/vyraz6/) or
        ...
        cuckoo.signature.hits(/vyraz7/) or
        cuckoo.signature.hits(/vyraz8/)
}
```

Spracovanie takéhoto množstva reportov nevyžaduje veľkú výpočtovú silu, a preto som ho vykonal na svojom osobnom počítači. Keďže nás zaujíma porovnanie dvoch verzií YARY, nepotrebujeme použiť najvýkonnejší server, keďže na konci dňa nás aj tak zaujíma porovnanie oboch verzií bežiacich na tom istom stroji v tej istej konfigurácii. Vďaka tomu nám na konci zostane vierohodné porovnanie dvoch skenovacích metód bez toho, aby vo výsledku hral rolu absolútny výkon stroja.

Pri testovaní ale dostávame rozporuplné výsledky. Na malej vzorke reportov dochádza pri mnohých inštanciách zrýchlenie o `1-10` násobok pôvodnej rýchlosti. Pri dlhšie trvajúcich skenoch inštancií ale dochádza k opačnému výsledku, kde sa výkon bajtkódu naopak spomalí `1-4` násobne.

Stávajúce riešenie [s]	Hyperscan [s]	Relatívny podiel [%]
2.22	2.77	124.55

Tabuľka 5.2: Celková doba vykonávania bajtkódu pri 100 reportoch. Porovnanie jednotlivých reportov nájdete v prílohe 1.

Z výsledkov badať, že Hyperscan napriek sľubným vlastnostiam nie len, že nedokáže predbehnúť stávajúce riešenie, ale dokonca za ním výrazne zaostáva. Jednou z pracovných hypotéz, čím to môže byť spôsobené, sa stal princíp skráteného vyhodnocovania popísaný v 4.1. Na jeho základe je možné predpokladať, že z celkového počtu skenov sa skutočne vykoná len istá časť. To by spôsobilo, že z porovnávania vyhodnocovania všetkých výrazov stávajúcim systémom alebo Hyperscanom by sa razom stala problematika porovnávania menšej časti výrazov stávajúcim systémom alebo porovnávania väčšej časti výrazov Hyperscanom. Z tohto dôvodu napriek prirodzene pomalšiemu skenovaniu stávajúceho systému by sa skenovanie dokončilo skôr, ako skenovanie Hyperscanom, a to z dôvodu menšieho počtu skenovaných údajov.

Stávajúce riešenie [s]	Hyperscan [s]	Relatívny podiel [%]
132.64	133.42	100.59

Tabuľka 5.3: Celková doba vykonávania bajtkódu pri 100 reportoch s vypnutým skráteným vyhodnocovaním. Porovnanie jednotlivých reportov nájdete v prílohe 2.

Z týchto dát môžeme vyčítať niekoľko vecí. Okrem tendencie dosahovať lepšie rýchlosti pri kratších skenoch a horšie rýchlosti pri dlhších skenoch sa tento trend potvrdzuje aj pri skenovaní s vypnutým skráteným vyhodnocovaním. To znamená, že rozdiel medzi medzi lepším a horším výsledkom Hyperscanu nespočíva v tom, kde sa ukončí vyhodnocovanie. V momente, kedy odoberieme z tejto rovnice skrátené vyhodnocovanie, jediným meniacim sa faktorom ostávajú samotné reporty, v ktorých sa v niektorých prípadoch skenuje jednou funkciou viac záznamov, inokedy menej. Javí sa, že práve toto spôsobuje problémy Hyperscanu, istý druh záznamov v reporte sa skenuje veľmi pomaly a kvôli skenovaniu týchto záznamov stráca Hyperscan inak výborné zrýchlenie z reportov, v ktorých sa problémových záznamov nachádza minimum, a teda nestihne stratiť svoju nadobudnutú rýchlostnú výhodu.

V reálnom svete sa ale nepoužívajú pravidlá v takej forme, akej sme ho prezentovali. Skutočné pravidlá obsahujú malý počet výrazov vzájomne prepojených vzťahmi `AND` a `OR`, ktoré vytvárajú medzi výrazmi tak nepredpovedateľné vzťahy, že takéto chovanie nie sme schopní emulovať naším testovacím pravidlom. A tak si ukážme, ako by prebiehalo skenovanie reportov skutočnými pravidlami, ktoré firma Avast skutočne využíva vo svojom produkčnom prostredí. Pre dôveryhodnejšie výsledky budeme prehľadávať 1000 reportov, keďže pri vyššom počte súborov sa lepšie ustáli rýchlostný pomer stávajúceho a hyperscanového riešenia, ktorých relatívna rýchlosť kolíše súboru od súboru, ako sme si ukázali pri analýze 3.

Stávajúce riešenie [s]	Hyperscan [s]	Relatívny podiel [%]
308.32	966.86	313.58

Tabuľka 5.4: Porovnanie fáz skenu rôznych verzií YARY na vzorke 1000 reportov.

Celkový počet skenov	Počet vykonaných skenov	Časť vykonaných skenov [%]
13 428 000	8 323 895	61.99

Tabuľka 5.5: Porovnanie počtu vykonaných skenov oproti celkovému počtu skenov v pravidlách.

Podľa očakávania teda k celkovému rýchlejšiemu vyhodnoteniu nedochadza. Musíme pristúpiť k manuálnej analýze behov inštancií YARY nad jednotlivými reportmi podobne, ako tomu bolo v ukážke 5.2. Pre to bolo vybratých prvých 20 inštancií behu. V doloženej prílohe 3 môžeme sledovať, že v mnohých inštanciách skutočne dochádza k miernemu zrýchleniu pozorovanému v niektorých skenoch nad vlastným pravidlom. Zároveň ale pri iných inštanciách dochádza k výraznému spomaleniu. Vďaka našim predchádzajúcim experimentom už vieme, že občasné drastické spomalenia skenov reportov nie sú náhodné. Predmetom

ďalšieho skúmania bude to, či by sa pozitívne odrazilo na výkone rozdelenie databáz pre regulárne výrazy a obyčajné textové výrazy, experimenty s rôznymi modifikátormi pri kompilácii databáz alebo zefektívnenie prehľadávania nájdených zhôd v texte. To je už ale nad rámec tejto práce, a tak musíme konštatovať, že aj napriek snahe dosahuje Hyperscan zrýchlenie len pri skenoch, ktoré zvláda pomerne rýchlo aj stávajúce riešenie. Naopak skeny, ktoré stávajúcemu riešeniu trvajú dlho, trvajú Hyperscanu ešte dlhšie a na týchto ojedinelých reportoch prichádza o všetku nadobudnutú rýchlostnú výhodu z ostatných skenov.

Kapitola 6

Záver

Cieľom práce bolo vylepšiť nástroj YARA o nové syntaktické konštrukty a pridať novú funkciu do niektorého z modulov. V prípade syntaktických konštruktov sa konkrétne jedná o operátor of, ktorému bola pridaná schopnosť pracovať s poľom pravdivostných hodnôt a zjednodušiť tak písanie pravidiel, ktoré doteraz využívali zbytočne veľké množstvo cyklov, ktoré zneprehľadňovali kód.

Druhým vylepšením sú interné premenné pre jednotlivé pravidlá. Tie v súčasnosti podporujú všetky základné dátové typy ako reťazce, celé čísla, desatinné čísla, boolovské hodnoty, ale napríklad aj linkovanie štruktúr používaných v moduloch k premenným.

Všetky doterajšie zmeny v YARE boli implementované aj do nástroja Yaramod a nástroj Yara Rules Toolchain bol upravený tak, aby dokázal transformovať pravidlo z tejto verzie YARY do verzie určenej pre iné organizácie, ktoré využívajú verejnú open-source verziu YARY.

Posledným vylepšením, ktoré sa týka modulov, je vylepšenie modulu Cuckoo. Táto zmena rieši skenovanie Cuckoo reportov a jeho zrýchlenie, ktoré sa malo dosiahnúť technológiou Hyperscan. Pri jeho testovaní sa ale ukázalo, že Hyperscan je rýchlejší pri skenoch, ktoré zvláda aj stávajúce riešenie, naopak skeny reportov, pri ktorých je stávajúce riešenie veľmi pomalé, dosahuje Hyperscan ešte horšie výsledky a tým prichádza o svoju výhodu.

Na základe ďalšieho testovania sme tak určili najlepší a najhorší možný scenár, faktory vedúce k tomuto výsledku a ďalšie oblasti, na ktoré sa môžeme pri ďalšom výskume zamerať, aby sme dosahovali lepšie rýchlosti.

Práve zrýchlenie skenovania reportov bude aj po odovzdaní práce objektom môjho ďalšieho záujmu, keďže mojím cieľom je dosiahnuť konzistentne vyššiu rýchlosť skenovania, než je tomu pri existujúcom riešení. Ďalší postup pozostáva z rozdelenia databáz pre regulárne výrazy a textové reťazce, testovanie ďalších modifikátorov pri kompilácii, optimalizácia prehľadávania už uložených zhôd, ukladanie databáz na disk a ich použitie pri neskoršom skenovaní, ale napríklad aj dokončenie chýbajúceho medzičlánku slúžiaceho na propagáciu chýb vzniknutých počas skenu do vyšších vrstiev YARY, ktorý chýbal aj v stávajúcej verzii YARY.

Literatúra

[1] Aho, M. *Fast pattern matching: an aid to bibliographic search. Communications of ACM* [online]. Október 1975 [cit. 2021-01-14]. Dostupné z: https://dl.acm.org/doi/10.1145/360825.360855.

[2] Cox, R. *Regular Expression Matching: the Virtual Machine Approach* [online]. [cit. 2021-01-14]. Dostupné z: https://swtch.com/~rsc/regexp/regexp2.html.

[3] Feldmann, A. e. a. *The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic* [online]. Október 2020 [cit. 2021-01-14]. Dostupné z: https://doi.org/10.1145/3419394.3423658.

[4] Intel. *Hyperscan* [online]. [cit. 2021-01-14]. Dostupné z: https://www.hyperscan.io/.

[5] Kowalczyk, K. *PE* [online]. [cit. 2021-01-14]. Dostupné z: https://blog.kowalczyk.info/articles/pefileformat.html.

[6] Morgan, S. *Cybercrime To Cost The World $10.5 Trillion Annually By 2025* [online]. November 2020 [cit. 2021-01-14]. Dostupné z: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/.

[7] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification* [online]. May 1995 [cit. 2021-01-14]. Dostupné z: https://refspecs.linuxfoundation.org/elf/elf.pdf/.

[8] Vega, M. *Internet of Things Statistics, Facts Predictions [2020's Update]* [online]. November 2020 [cit. 2021-01-14]. Dostupné z: https://review42.com/internet-of-things-stats/.

[9] VirusTotal. *YARA* [online]. [cit. 2021-01-14]. Dostupné z: https://yara.readthedocs.io/.

Číslo reportu	Stávajúce riešenie [s]	Hyperscan [s]	Relatívny podiel [%]
1	0.023822	0.025966	109.00
2	0.030863	0.007245	23.47
3	0.007967	0.005691	71.43
4	0.027266	0.010334	37.90
5	0.003460	0.005407	156.27
6	0.009533	0.021357	224.03
7	0.003392	0.010530	310.44
8	0.073586	0.077400	105.18
9	0.029344	0.006626	22.58
10	0.003345	0.004170	124.66
11	0.026416	0.043631	165.17
12	0.007791	0.004326	55.53
13	0.124488	0.016496	13.25
14	0.112277	0.108318	96.47
15	0.002348	0.004088	174.11
16	0.008633	0.012959	150.11
17	0.002530	0.004894	193.43
18	0.019939	0.006592	33.06
19	0.002176	0.003771	173.30
20	0.002514	0.004887	194.39

Tabuľka 1: Porovnanie skenovania prvých 20 reportov.

Číslo reportu	Stávajúce riešenie [s]	Hyperscan [s]	Relatívny podiel [%]
1	1.918987	0.636932	33.19
2	0.034168	0.008033	23.51
3	0.035442	0.009206	25.97
4	0.082790	0.016057	19.39
5	0.113366	0.015477	13.65
6	0.262766	0.043253	16.46
7	0.112615	0.016780	14.90
8	10.899703	10.695194	98.12
9	0.034198	0.008301	24.27
10	0.051689	0.009068	17.54
11	2.196265	0.656523	29.89
12	0.008743	0.004557	52.12
13	0.131451	0.017056	12.98
14	8.782014	16.404565	186.80
15	0.033019	0.009734	29.48
16	0.223098	0.034093	15.28
17	0.032540	0.008734	26.84
18	0.393046	0.077332	19.68
19	0.027086	0.007143	26.37
20	0.031260	0.008397	26.86

Tabuľka 2: Porovnanie skenovania prvých 20 reportov bez skráteného vyhodnocovania.

Číslo reportu	Stávajúce riešenie [s]	Hyperscan [s]	Relatívny podiel [%]
1	0.204811	0.392304	191.54
2	0.012311	0.009172	74.50
3	0.011368	0.009976	87.76
4	0.013192	0.013528	102.55
5	0.015645	0.014651	93.65
6	0.049530	0.040193	81.15
7	0.014154	0.013873	98.01
8	2.659707	20.480034	770.01
9	0.008362	0.009920	118.63
10	0.009452	0.009915	104.90
11	0.189977	0.345147	181.68
12	0.006034	0.007441	123.32
13	0.017269	0.014794	85.67
14	1.047168	9.609828	917.70
15	0.013840	0.009991	72.19
16	0.031300	0.027304	87.23
17	0.008955	0.008590	95.92
18	0.061549	0.054587	88.69
19	0.008028	0.007040	87.69
20	0.008440	0.008670	102.73

Tabuľka 3: Porovnanie skenovania prvých 20 reportov bez skráteného vyhodnocovania.