VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYSOCE VÝKONNÁ PLATFORMA PRO ÚČELY VÝZKUMU MALWARU

HIGH-PERFORMANCE PLATFORM FOR MALWARE RESEARCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PAVOL PLASKOŇ**
AUTHOR

VEDOUCÍ PRÁCE — **Doc. Dr. Ing. DUŠAN KOLÁŘ**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Plaskoň Pavol, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Vysoce výkonná platforma pro účely výzkumu malwaru**
High-Performance Platform for Malware Research
Kategorie: Data mining

Zadání:

1. Studujte oblast vysoce výkonných a škálovatelných systémů se zaměřením na systémy využívající DBS.
2. Seznamte se se službami využívanými v rámci analýzy škodlivého kódu ve společnosti Avast. Zaměřte se na službu provádějící shlukovou analýzu a klasifikaci souborů Clusty - bude primárním zdrojem pro vytvářenou platformu.
3. Navrhněte systém, jenž bude přijímat data z existujících služeb, bude je agregovat a poskytne rozhraní pro jejich analýzu. Jde o označování vstupů (soubory, detekční definice atd.) pomocí tzv. tagů (severita, typ malwaru, jméno rodiny atd.) a dotazování nad nimi. Vstupní data je nutné prioritizovat, porovnávat a rozhodovat o výsledku. Cílem je vysoká výkonnost a škálování celého řešení.
4. Navržený systém implementujte. Rovněž navrhněte, implementujte a otestujte potřebná rozšíření služby Clusty (lepší autom. klasifikace, rozšíření API atd.).
5. Vytvořené řešení důkladně otestujte sadou jednotkových a integračních testů.
6. Svou práci zhodnoťte a diskutujte možný budoucí vývoj.

Literatura:

- T. Sterling, M. Anderson, M. Brodowicz: High Performance Computing: Modern Systems and Practices, Morgan Kaufmann (2017), ISBN 978-0124201583
- M. Sikorski, A. Honig: Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software, No Starch Press (2012), ISBN 978-1593272906
- Interní dokumentace společnosti Avast.
- Dále dle doporučení vedoucího či konzultanta.

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání a rozpracování čtvrtého bodu.

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

Vedoucí práce: **Kolář Dušan, doc. Dr. Ing.**
Konzultant: Zemek Petr, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 24. října 2018

Abstrakt

V antivírusových firmách sa denne analyzuje veľké množstvo súborov. Pre podporu ich analýzy a klasifikácie sa používajú rôzne automatizované nástroje. Detekčné definície pre detekciu a zablokovanie škodlivých programov sú taktiež už často generované rôznymi algoritmami. Informácie o aktuálne sa šíriacom malvéri sú teda roztrúsené v niekoľkých nástrojoch a niektoré sú príliš generické. Táto práca popisuje návrh nového systému, ktorý bude všetky dostupné informácie agregovať, prioritizovať a vyhodnocovať. Kvôli veľkému množstvu vstupných dát bude kritickou časťou výkonnosť a škálovateľnosť celého systému. Súbory, detekcie a prípadne ďalšie objekty budú označované pomocou tzv. tagu, ktorý vyberie alebo odvodí z dostupných znalostí od analytických nástrojov. Nad uloženými informáciami bude poskytovať rozhranie pre ich ďalšie spracovanie či generovanie štatistík. Navrhnutá platforma bola implementovaná, otestovaná a nasadená do produkcie.

Abstract

Anti-malware companies analyze large number of files every day. In order to speed up their analysis, many automatized tools were implemented. Detection definitions that detect malicious software are often generated automatically. Information about currently spreading malware is scattered across several tools and they are sometimes too generic. This work proposes a new tool that will aggregate, prioritize, and evaluate all the available information. Due to large amount of incoming data, high performance and scalability of the system is necessary. Files, detection definitions, and other objects will be tagged using the given information directly or inferred. Collected information will be accessible via interface for further analysis and statistics. Everything was implemented, tested and put into production.

Kľúčové slová

analýza malvéru, detekčné definície, klasifikácia, škálovateľnosť, tag

Keywords

malware analysis, detection definitions, classification, scalability, tagging

Citácia

PLASKOŇ, Pavol. *Vysoce výkonná platforma pro účely výzkumu malwaru.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Dr. Ing. Dušan Kolář

Vysoce výkonná platforma pro účely výzkumu malwaru

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Doc. Dr. Ing. Dušana Koláře. Ďalšie informácie mi poskytli Petr Zemek a Jakub Křoustek. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Pavol Plaskoň
20. mája 2019

Poďakovanie

Rád by som poďakoval svojmu vedúcemu Doc. Dr. Ing. Dušanovi Kolářovi a hlavne Ing. Petrovi Zemkovi a Ing. Jakubovi Křoustkovi za konzultácie, spätnú väzbu a odborné rady.

Obsah

Kapitola 1

Úvod

S rastúcim množstvom počítačov a elektronických zariadení narastá aj množstvo šíriacich sa škodlivých programov, tzv. *malvér*. Medzi najčastejšie ciele patria najrozšírenejšie operačné systémy pre osobné využitie, teda Windows a Android [10]. S rozvojom internetu vecí (angl. *Internet of Things*, IoT) sa rýchlo začína rozširovať aj škodlivý softvér a útoky na tieto zariadenia. Podľa [33] narástol počet útokov na IoT o 600 % za posledný rok.

Denne sa do antivírusových firiem dostávajú státisíce súborov (vzoriek) na analýzu [12]. Množstvo zachytených a zablokovaných súborov u koncových užívateľov je ešte vyššie. Pri takýchto číslach je už nevyhnutnosťou používať automatizované nástroje na detekciu škodlivých programov a nástroje na generovanie detekčných definícií (skrátene, detekcií). Detekčnú definíciu môžeme zjednodušene chápať ako súhrn definovaných podmienok, pravidiel, ktoré musí daný súbor spĺňať. V prípade detekcie súboru danou definíciou ho môžeme zaradiť do kategórie programov, ktoré daná detekčná definícia popisuje. Samozrejme, je možné vytvoriť si detekciu aj pre neškodlivé súbory, osobné dokumenty a pod., ale v tejto práci budeme uvažovať len detekcie škodlivých programov.

Pri tak veľkej diverzite škodlivých programov sa zaužíval systém pre ich rozlišovanie, kategorizovanie. Na najvyššej úrovni môžeme hovoriť o miere škodlivosti programu. Zatiaľ bol spomenutý len malvér ako škodlivý program. Delenie na škodlivé a neškodlivé programy ale nie je dostatočné. Objavuje sa aj veľké množstvo tzv. potenciálne škodlivých programov (angl. *Potentially Unwanted Program*, PUP). Tie sa nesnažia priamo poškodiť svoj cieľ, len ho nejako využiť pre svoje zámery, napríklad pre šírenie reklamy pomocou bannera vo webovom prehliadači. Ďalej sú tu napríklad programy na prelomenie ochrany iných programov, hlavne počítačových hier, proti spusteniu bez licencie, dolovanie kryptomien a i. V tomto prípade nejde o priame poškodenie užívateľovho počítača, či kradnutie dát, ako sa to spája väčšinou s pojmom malvér, ale taktiež je tam istý druh poškodzovania, zneužívania zdrojov.

Na nižšej úrovni sa ďalej delia programy podľa rôznych charakteristík na typy, napríklad trojan, ransomware, červ. Či už ide o ich spôsob šírenia, ukrývanie sa, vykonávanú činnosť a i. Pre rozlíšenie programov v rámci typov sa už používa len rozdelenie do rodín/kmeňov. Ide o analytikmi vytvorené pomenovania skupín s takmer identickými programami. Rôzne varianty jedného programu v prípade výraznejšej črty sa ešte môžu ďalej označiť ako konkrétna varianta danej rodiny, čo je len vlastne ďalší názov vytvorený analytikmi. Pre označenie programu triedou škodlivosti, typom, rodinou, prípadne variantou a cieľovou platformou sa bude v rámci tejto práce používať ďalej len pojem *tag* (značka).

Už pri prvých vírusoch sa zaužíval zvyk označovať ich názvami. Termín *vírus* nie je v tomto prípade ale úplne jednoznačný. Väčšina zdrojov o prvých škodlivých programoch

ich označuje ako vírusy, ale neskôr sa začali škodlivé programy detailnejšie rozdeľovať na rôzne typy, podľa istých charakteristík. Tomuto deleniu sa podrobnejšie venuje sekcia 3.3. Napríklad program Creeper sa často označuje ako vírus. Vytvoril ho Bob Thomas v roku 1971 a jeho účelom bolo len pohybovať sa v ARPANET sieti. Neskôr bol vylepšený a vedel sa kopírovať medzi počítačmi, namiesto presúvania sa. Takéto správanie sa dnes označuje ako *červ* (angl. *worm*). Tento termín zaviedli John Shoch a Jon Hupp až v roku 1979 [2]. Teda termín vírus sa aj dnes používa často ako obecné označenie škodlivého programu, ale v kontexte analýzy škodlivých programov ide len o označenie jedného z mnohých typov malvéru.

Okrem automatizovaných nástrojov sa manuálnej analýze škodlivých programov a tvorbe detekčných definícií neustále venuje veľká pozornosť. Zaradenie do triedy škodlivosti, typu a rodiny môže byť ešte stále presnejšie z manuálnej analýzy, ale časovo je to náročné. Pri automatickom generovaní detekčných definícii vzniká často veľké množstvo generických pravidiel, ktoré pokrývajú niekedy aj viacero rodín, prípadne typov a stráca sa takto informačná hodnota. Jediná informácia je, že je to škodlivý program. Samozrejme, vyvíjajú sa aj nástroje, ktoré sa snažia všetky tieto informácie o programoch podchytiť a zachovať. Podrobnejšie sa nástrojom vyvíjaným v spoločnosti Avast Software venuje kapitola 3. Takýchto nástrojov je viacero a informácie použité pri vytváraní tagov pre súbory sa líšia. Niektoré sú postavené na dynamickom správaní programu, iné na statickom, prípadne ich kombinácii. Niektoré z nich sú spoľahlivejšie, iné menej.

Správny tag programu sa dá využiť napríklad pre spresnenie tagu pre detekčnú definíciu, za predpokladu, že pokrýva jednu rodinu. Ďalej sa môže použiť pre odvodenie tagu pre indikátory škodlivej činnosti (angl. *Indicator of Compromise*, IoC), kedy daná rodina malvéru vykazuje určité špecifické znaky, ako je pripájanie sa na istú adresu v sieti, vytváranie, či modifikovanie špecifických systémových artefaktov (registre, súbory, pomenované objekty a i.). Podľa tagu môžeme ďalej vygenerovať štatistiky šírenia sa škodlivých programov vo svete, či už ide o počty útokov v jednotlivých krajinách, šírenie sa v jednotlivých dňoch istého časového intervalu, zistiť aktuálne najrozšírenejšiu rodinu, a i.

Všetky tieto informácie sú roztrúsené v jednotlivých nástrojoch a ich prepojenie napríklad za účelom lepšej analýzy danej rodiny vzoriek by si vyžadovalo veľa manuálnej práce od analytika. Cieľom tejto práce je analyzovať analytické nástroje v spoločnosti Avast, ktoré môžu byť užitočné pre získavanie týchto informácií. Ďalej bude navrhnutá platforma, ktorá bude agregovať všetky tieto dostupné informácie o tagoch pre súbory, detekčné definície, prípadne informácie o indikátoroch narušenia v programoch. Získané informácie bude potrebné vyhodnocovať a prepájať, pretože k dispozícii je väčšinou len tag pre programy. Prínosom môže byť hlavne tagovanie detekčných definícií a IoC, čo môže následne vyústiť v lepšie špecifikované rodiny.

Vzhľadom na veľké množstvo denne prichádzajúcich vzoriek môžeme predpokladať ešte väčšie množstvo informácií poskytovaných od analytických nástrojov a bude potrebné sa taktiež zamerať na dostatočnú výkonnosť a škálovateľnosť navrhovanej platformy.

Práca je rozdelená nasledovne. Vysokovýkonnými systémami sa zaoberá kapitola 2 a ide hlavne o zameranie na databázové systémy, pretože agregované informácie je nutné niekde ukladať a mať ich stále k dispozícii. V kapitole 3 sú predstavené analytické nástroje spoločnosti Avast so zameraním na nástroj Clusty. Návrh platformy je popísaný v kapitole 4. Návrh, implementácia a testovanie rozšírení nástroja Clusty sú popísané v kapitole 5. Implementácia platformy je uvedená v kapitole 6. Testovanie platformy je popísané v kapitole 7. Zhrnutie tejto práce a ďalšie pokračovanie je v poslednej kapitole 8.

Kapitola 2

Vysoko výkonné a škálovateľné systémy

Problematika vysokovýkonného počítania patrí k multidisciplinárnym oborom. Kombinuje hardvérové technológie a architektúru s operačnými systémami, programovacími nástrojmi, softvérom, algoritmami a riešeným problémom. Rozvojom všetkých týchto oblastí sa s postupom času rapídne urýchľujú aj náročné výpočty. Z tisíc základných operácii za sekundu v štyridsiatych rokoch dvadsiateho storočia sa dnes dostávame na sto kvadriliónov floating-point operácii (100 petaflopov) za sekundu u superpočítačov [32].

Meranie výkonnosti sa ale netýka len počtu operácií za sekundu. Metrík je viacero a žiadna nezahŕňa všetky aspekty kvality výpočtu. Medzi základné metriky patrí už spomínaný čas a počet vykonaných operácií za daných podmienok. Pri superpočítačoch sa typicky hovorí o počte floating-point operácii za sekundu, tzv. *flops*. V prípade riešenia reálnych problémov je zaujímavejšou metrikou len samotný čas výpočtu. Vzhľadom na ich obrovský počet je ale obtiažne hodnotiť výkonnosť s použitím tejto jednotky. Preto sa používa istá skupina štandardizovaných problémov, tzv. *benchmarks*, pomocou ktorých môžeme porovnávať doby výpočtu na dvoch nezávislých systémoch pri riešení rovnakého problému.

Urýchľovanie výpočtov na superpočítačoch je rozsiahla téma, zahŕňajúca optimalizácie na úrovni procesorových inštrukcií, architektúry procesorov – zvyšovanie počtu jadier, vlákien, vektorové procesory, práce s pamäťou, cache pamäťami (dátové, inštrukčné), optimalizácie zdrojového kódu prekladačmi, paralelné výpočty – zdieľaná pamäť, zasielanie správ, atď. Detailne sa tomuto venuje [4, 8].

Okrem problematiky superpočítačov sa požiadavky na vysokú výkonnosť týkajú v dnešnej dobe aj databázových systémov a spracovávania obrovského množstva dát. V tejto oblasti vstupujú do hry hlavne diskové operácie, replikácia dát, vyvažovanie záťaže, či aj sieťová komunikácia. SQL databázy stručne popisuje sekcia 2.1, NoSQL databázy sú predstavené v sekcii 2.2. Problematike ich škálovateľnosti sa podrobnejšie venuje sekcia 2.3. Sekcia 2.4 sa venuje porovnaniu týchto dvoch typov databázových systémov, SQL a NoSQL, z hľadiska ich výkonnosti.

Ďalším typom vysokovýkonných systémov sú aj distribuované systémy, ktoré ale na rozdiel od superpočítačov sú asynchronné a zamerané na iné typy úloh. Podrobnejšie budú popísané v sekcii 2.5.

2.1 SQL databázové systémy

Tradičné databázové systémy sú postavené na relačnom modeli dát. Označované sú aj SQL databázy kvôli používaniu SQL jazyka [41]. Patrí sem napríklad PostgreSQL, MySQL, Microsoft SQL Server . . .

Hlavnými znakmi relačných databáz sú [35]:

- Postavené na matematickom modeli.
- Data sú uložené v tabuľkách.
- Normalizácia dát – je zavedených niekoľko normálnych foriem, ktorých hlavným cieľom je redukovať neatomické atribúty a znížiť redundanciu v dátach. Obecne je databáza považovaná za normalizovanú, ak je v tretej normálnej forme.
- Integritné obmedzenia – primárny, cudzí kľúč.
- ACID transakčný model – vyžaduje tieto štyri vlastnosti od transakcií:
  - atomicita – ak zlyhá nejaká časť transakcie, tak je celá transakcia neplatná, teda *všetko alebo nič*.
  - konzistencia – zaisťuje stabilnosť a validitu databázy pred a po vykonaní transakcie.
  - izolácia – viacero súbežných transakcií sa nijako neovplyvňuje. Po ich vykonaní je stav databázy rovnaký, akoby boli spustené sériovo.
  - trvanlivosť – po úspešnom zapísaní transakcie sa stav databázy uchová aj v prípade výpadku.
- Konzistencia má prednosť pred výkonom a dostupnosťou.
- Škálovateľnosť – tieto systémy využívajú hlavne vertikálnu škálovateľnosť. Horizontálna (a obecne distribuované relačné databázy) sú náročnejšie na udržanie ACID a integrity dát, spájanie tabuliek medzi systémami je taktiež náročné [17].
- Komplexné SQL dotazy.
- Striktná schéma dát.
- Vhodné pre štruktúrované dáta.

PostgreSQL

PostgreSQL je objektovo-relačný databázový systém, ktorý používa a rozširuje jazyk SQL. Získal si veľkú popularitu hlavne pre svoju spoľahlivosť, dátovú integritu, veľké množstvo robustných funkcionalít, rozšíriteľnosť, odolnosť voči chybám či podporu ACID od roku 2001 [34]. Medzi ďalšie významné vlastnosti patrí:

- Široká podpora dátových typov, zahŕňajúca aj dokumenty (JSON, XML), geometrické objekty a umožňuje definovať vlastné typy.
- Dátová integrita – podpora obmedzení a zamykania.
- Paralelizmus a vysoký výkon, indexovanie, transakcie a vnorené transakcie.
- Spoľahlivosť a zotavenie sa po chybách – Write-Ahead Logging (WAL), replikácia – synchronná, asynchronná, logická.

- Bezpečnosť – autentifikácia, zabezpečenie na úrovni riadkov a stĺpcov.
- Rozšíriteľnosť – uložené funkcie a procedúry, podpora procedurálnych jazykov, priestorové rozšírenie PostGIS[1].

Replikácia zabezpečuje vysokú dostupnosť a vyvažovanie záťaže. V prípade výpadku primárneho serveru ho môže okamžite nahradiť jeden zo sekundárnych serverov (tzv. *standby* servery).

Streaming replikácia v PostgreSQL je postavená na zasielaní a aplikovaní WAL z primárnej databázy na sekundárne. WAL je technika zabezpečujúca dátovú integritu tak, že každá zmena dát musí byť najprv zapísaná do logovacieho súboru na disku. Až potom môžu byť dáta pozmenené v databáze. V prípade poruchy je možné obnoviť databázu podľa WAL logov. Takto sa redukuje počet zápisov na disk, pretože pri každej transakcii sa musí na disk zapísať len logovací súbor, ktorý je sekvenčný [34].

Natívne sa používa asynchronná replikácia, pri ktorej môže nastať malé oneskorenie medzi zápisom dát v primárnej databáze a ich reflektovaním v sekundárnych.

V prípade zlyhania primárneho serveru sa niektoré transakcie ešte nemuseli dostať do sekundárnych a dôjde k strate dát. Tento problém sa snaží riešiť synchronná replikácia, ktorá vyžaduje od sekundárnych serverov potvrdenie, že daná transakcia bola na nich úspešne zapísaná. Až po úspešnom zápise na všetkých serveroch je považovaná za kompletnú. Zvyšuje sa tak trvanlivosť dát, ale narastá aj čas potrebný pre vykonanie transakcie.

2.2 NoSQL databázové systémy

Ide o nerelačné databázové systémy, ktoré už nepoužívajú jazyk SQL. Patrí sem napríklad HBase, Cassandra, Redis, MongoDB.

Hlavnými znakmi nerelačných databáz sú [35]:

- Data uložené ako JSON dokumenty, dvojice kľúč hodnota, grafy, stĺpce ai.
- Denormalizové uloženie dát, všetky data pre jeden objekt sú typicky v jednom zázname.
- Vhodnejšie pre komplexné a viacúrovňové dáta.
- Škálovateľnosť je typicky horizontálna a podporujú beh na veľkom množstve uzlov.
- ACID model je v NoSQL nahradený BASE modelom, ktorý je definovaný nasledovne: angl. *Basically Available, Soft state, Eventual consistency*. Ide teda o garanciu prakticky neustálej dostupnosti. Na druhej strane ale databáza nemusí byť konzistentná neustále, ale v určitých momentoch bude zaručená aj konzistencia [22].
- CAP teorém – tvrdí, že pre distribuované uloženie dát môžeme garantovať maximálne dve z týchto troch vlastností – konzistencia (ako v ACID), dostupnosť, odolnosť v prípade rozpadu časti siete.
- Konzistencia, dostupnosť a výkon sa teda môžu meniť jedno na úkor iných podľa potrieb aplikácie (napr. zvýšenie výkonu na úkor konzistencie).
- Komplexné operácie sú časovo náročnejšie.

[1] https://postgis.net/

Cassandra

Cassandra je vysokoškálovateľná distribuovaná NoSQL databáza navrhnutá pre zvládnutie veľkého množstva štruktúrovaných aj neštruktúrovaných dát.

Medzi významné vlastnosti tejto databázy patrí [3]:

- Horizontálna škálovateľnosť na veľké množstvo uzlov. Uzly sú si rovnocenné – tzv. *peer-to-peer* architektúra (nerozlišuje sa teda *master* vs. *slave*).
- Lineárna škálovateľnosť – zvyšuje sa výkon s pridanými uzlami.
- Jednoduchá automatická distribúcia dát a replikácia.
- Nemá kritické miesto, na ktorom závisí dostupnosť celej databázy (tzv. *single point of failure*).
- Rýchly zápis dát.
- Z ACID vlastností ponúka AID – teda atomicitu, izoláciu a trvanlivosť dát. Mieru konzistencie si môže nastaviť administrátor.
- Štruktúra uloženia dát vychádza z dotazov nad nimi, nie zo štruktúry dát. To vedie na duplicity v dátach, chýbajúcu agregáciu, spájanie dát a vnorené dotazy v dotazoch.

Využitie nachádza hlavne v real-time aplikáciách, sociálnych sieťach, real-time dátových analýzach, službách s veľkým množstvom dát, množstvom zápisov atď.

MongoDB

NoSQL databáza, ktorá ukladá dáta v BSON dokumentoch, čo je vlastne binárna reprezentácia JSON dokumentov.

Základné vlastnosti [23]:

- Dokumenty nemajú definovanú schému (oproti tabuľkám v relačných databázach).
- Limitovaná veľkosť dokumentu a limitovaná úroveň vnorených dokumentov.
- Podporuje horizontálnu škálovateľnosť a distribúciu dát.
- Replikácia a vysoká dostupnosť.
- Vlastný dotazovací jazyk postavený nad dokumentmi s podporou dynamických dotazov.
- Chýbajú komplexné spájania dát (angl. *join* operácie).

2.3 Škálovateľnosť databázových systémov

Hlavným cieľom škálovateľnosti a elasticity databázových systémov je zlepšiť ich dostupnosť a výkon, najmä v prípade meniacich sa nárokov na ich využitie. Táto časť vychádza z [11] a [24].

Škálovateľnosť je schopnosť systému vysporiadať sa s narastajúcim množstvom práce. Teda systém musí zvládnuť viac práce za rovnaký čas a zvýšiť priepustnosť pridaním ďalších výpočtových prostriedkov. S tým súvisí aj dostupnosť databázového systému a možnosť vykonávať administratívne úkony a úpravy (napr. zmena schémy) bez značného vplyvu na aplikácie a dostupnosť pre koncových užívateľov [24].

Elasticita označuje stupeň do akej miery je systém schopný adaptovať sa na zmeny záťaže poskytovaním ďalších prostriedkov, či ich odoberaním na základe aktuálnej potreby. V každom časovom okamihu je teda pri danej záťaži k dispozícii takmer adekvátne množstvo prostriedkov. Cieľom je teda mať k dispozícii len taký počet prostriedkov, aký je aktuálne potrebný, vyhnúť sa nedostatku alebo prebytku. Nedostatok môže vyústiť v problémy s výkonom a dostupnosťou, a má teda priamy vplyv na koncového užívateľa. Prebytok zase spôsobuje zbytočnú spotrebu zdrojov a energie, čo prináša vyššie finančné náklady. Vyššia elasticita teda znamená lepšiu schopnosť tolerovať zmeny a vyššiu mieru jednoduchosti pri ich aplikovaní [24].

Škálovateľnosť sa týka ako klasických relačných databázových systémov, tak aj postrelačných NoSQL systémov. Tie získali na popularite aj vďaka ich veľmi dobrej škálovateľnosti a dostupnosti. Tieto databázy zjednodušujú dizajn a obetujú tradičný ACID model pre vyšší výkon a dostupnosť pre obrovské množstvo dát.

Škálovateľnosť databázových systémov môžeme rozdeliť obecne na dve kategórie. Oba popísané prístupy je možné kombinovať.

- vertikálna – v tomto prístupe ide o pridávanie výpočtových zdrojov (operačná pamäť, procesory) do existujúceho systému (tzv. *scaling up*, pri odstraňovaní zdrojov ide o tzv. *scaling down*). Výsledkom je samozrejme nárast výkonu aj keď za cenu nedostupnosti systému počas pridávania nových zdrojov. Taktiež rozširovanie zdrojov jedného systému má svoje limity, hardvérové aj softvérové (operačný systém). V prípade uloženia všetkých dát na jednom systéme nezaručuje vertikálne škálovanie dostupnosť dát v prípade výpadku systému.

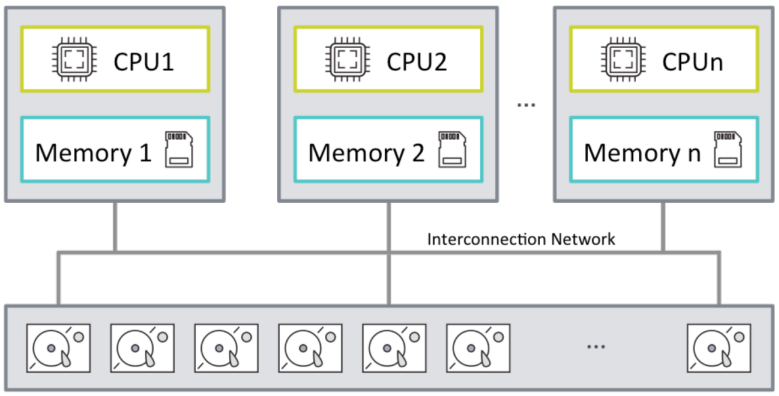
- horizontálna – označovaná ako *scaling out*, kedy sa pridáva ďalší hardvér, teda celé výpočtové jednotky (servery) k systému (pri odstraňovaní hardvéru ide o tzv. *scaling in*). Tento typ škálovateľnosti ale môže byť zase limitovaný možnosťami softvéru (napr. rýchlosťou sieťovej komunikácie). U tradičných databázových systémov hovoríme len o škálovateľnosti na niekoľkých strojoch, nejde o žiadne stonásobné zvýšenie. Oproti vertikálnemu taktiež narastá komplexnosť spravovania a môže vznikať oneskorenie pri komunikácii uzlov.

2.3.1 Metódy

Pre škálovanie databáz sa väčšinou využíva cluster výpočtových jednotiek. Všetky zapojené servery sa starajú o databázové požiadavky. Dominantné sú tieto architektúry:

- *shared-disk* – ide o typ horizontálneho škálovania. Všetky spojené systémy zdieľajú jedno diskové pole. Každý procesor má svoju vlastnú operačnú pamäť, ale všetky pristupujú priamo ku spoločným diskom, viď obrázok 2.1. Dáta nie je nutné rozdeľovať medzi systémami. Každý procesor má vlastnú cache pre dáta a je potrebné zaistiť konzistenciu v prípade modifikácie dát viacerými jednotkami. Medzi hlavné výhody tohto prístupu patrí jednoduchá rozšíriteľnosť, dostupnosť, rozdelenie záťaže a migrácia z centralizovaného systému.

- *shared-nothing* – ide o typ horizontálneho škálovania. Každý systém má vlastnú operačnú pamäť a disky. Procesory komunikujú navzájom pomocou sieťového prepojenia, viď obrázok 2.2. Požiadavky od klientov sú automaticky smerované na systém, ktorý má daný zdroj. Vždy len jeden systém vlastní daný zdroj. Hlavnou výhodou tohto systému je veľmi dobrá škálovateľnosť. Teoreticky môže ísť až o tisíce procesorov,



Obr. 2.1: Ukážka architektúry *shared-disk* pre škálovanie DB systému (prevzatý z [24]).

pretože sa navzájom neovplyvňujú, nič nezdieľajú. Prakticky je to číslo ale výrazne nižšie [24]. Tento prístup je vhodný pre systémy s veľkým množstvom operácií čítania, analytické spracovanie, napríklad pre dátové sklady.

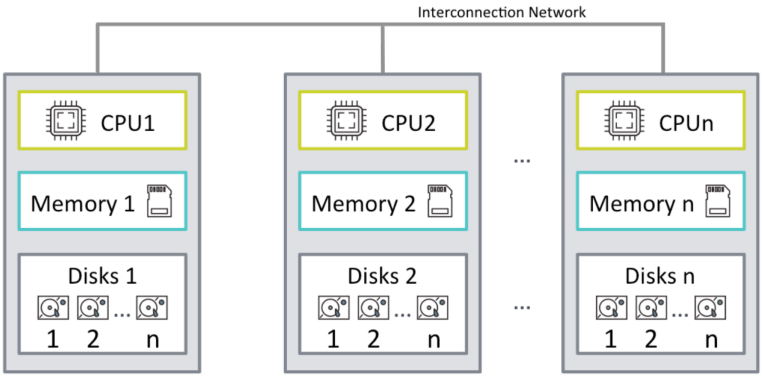
Nevýhodou môže byť nutnosť navrhnutia schémy tak, aby bolo možné rozdeliť dáta medzi uzly. Dáta nie sú zdieľané a sú teda uchované len na jednom uzle. Databázový systém musí vedieť dáta rozdeliť a vedieť ako k nim pristúpiť. Väčšinou sa používa horizontálne rozdelenie, kedy sa určia stĺpce, podľa ktorých sa rozdelí tabuľka na viacero tabuliek. Každá bude obsahovať inú podmnožinu riadkov pôvodnej tabuľky. V jednoduchom prípade môže ísť o rozdelenie tabuľky napr. na údaje z Európy a údaje z Ázie. V prípade komplikovanejších schém dát, kde to nejde takto jednoducho rozdeliť, sa môže použiť komplikovanejšie rozdelenie, napr. podľa hash kľúča. Prístup k dátam na viacerých uzloch naraz, prípadne ich modifikácia sú obtiažnejšie a môžu sa narušiť ACID vlastnosti databázy.

Automatické rozdelenie sa označuje *sharding*, viď nižšie. Ďalšou možnosťou je vertikálne delenie, teda stĺpce tabuľky sú rozdelené na podmnožiny.

Tento prístup je implementovaný napríklad v Teradata, MySQL, či niektorých NoSQL databázach. Efektívne to môže byť najmä v NoSQL, kde nie je kladený taký dôraz na ACID.

- *sharding* – táto technika sa často používa so *shared-nothing* prístupom pre automatické rozdelenie dát a ich spravovanie. Dáta sú rozdelené na menšie časti (*shards*). Ide o špeciálne horizontálne rozdelenie tabuliek podľa riadkov, ale jednotlivé tabuľky už nemusia byť nutne na jednom databázovom serveri. Hlavnou výhodou je teda rozdelenie záťaže na viacero serverov.

  Jednou z nevýhod tohto prístupu je nutnosť spravovať viacero osobitných databázových serverov. Ďalej to môže mať negatívny dopad na odolnosť voči poruchám, kedy pri zlyhaní jedného serveru sú jeho dáta nedostupné. Preto sa často dopĺňa tento prístup replikáciou (viď sekciu 2.3.2).



Obr. 2.2: Ukážka architektúry *shared-nothing* pre škálovanie DB systému (prevzatý z [24]).

Medzi databázové systémy, ktoré implementujú *sharding* patrí napríklad Couchbase, Apache HBase, a MongoDB.

2.3.2 Replikácia dát

Ide o redundantné ukladanie dát pre zvýšenie odolnosti voči chybám, kedy je kópia dát uložená na jednom alebo viacerých ďalších uzloch. Kópie dát je ale potrebné vždy udržiavať aktualizované s originálom. V prípade zmeny v hlavnej databáze musí systém zaručiť aktualizáciu dát na všetkých kópiách.

Z pohľadu miesta aktualizácie dát pri replikácii môžeme použiť tieto architektúry [26]:

- *master-slave* replikácia – centralizovaná architektúra, ktorá umožňuje aktualizáciu dát len na jednom (master) uzle. Všetky zmeny musia byť následne propagované do ostatných uzlov. Výhodou je jednoduchá aktualizácia, ale jediný master uzol je zároveň kritickým miestom systému.
- *master-master* (multi-master) replikácia – požiadavka na aktualizáciu dát môže byť zaslaná na ktorýkoľvek master uzol. Aktualizácie môžu byť na rôznych uzloch v rôznom poradí a synchronizácia dát je komplikovanejšia.

Z pohľadu synchronizácie dát medzi uzlami môžeme využiť tieto prístupy [7]:

- *synchronná* (angl. aj *eager*) – zmeny musia byť rozposlané a potvrdené na všetkých uzloch. Až potom môže uzol, na ktorý bola zaslaná požiadavka na aktualizáciu, či zápis dát, potvrdiť ich úspešné zapísanie do databázy. Predlžuje to teda čas vykonania zmien, ale zvyšuje to konzistenciu medzi replikami.
- *asynchronná* (angl. aj lazy) – uzol, ktorý prijal požiadavku na aktualizáciu alebo zápis, aplikuje dané zmeny, potvrdí ich a asynchronne ich zašle na zvyšné uzly, ale nečaká už na ich potvrdenie. Zvyšuje sa takto rýchlosť zápisu, ale môže nastať nekonzistencia dát medzi replikami (napr. v prípade výpadku uzlu).

Všetky štyri kombinácie podľa miesta aktualizácie a spôsobu synchronizácie sú prakticky realizovateľné. U distribuovaných relačných systémov je väčšinou preferovaná synchronná master-slave replikácia pre udržanie konzistencie. NoSQL databázy typicky používajú

asynchronnú replikáciu v kombinácii s master-slave (napr. databázy MongoDB, HBase) aj master-master architektúrou (napr. databázy Cassandra, Dynamo) [7].

SQL servery podporujú niekoľko typov replikácií z hľadiska spôsobu synchronizácie dát [20]:

- *snapshot* replikácia – vytvára sa identická kópia dát pri každom spustení, tak ako sú dáta k dispozícii v danom momente. Nie je k dispozícii žiadna nasledovná synchronizácia a ďalšie spustenie replikácie kopíruje znova celú databázu, aj bez ohľadu na prípadne externé zmeny v cieľovej databáze. Používa sa typicky pri počiatočnom vytvorení replikácie a následne sa pre synchronizáciu už používajú metódy uvedené nižšie.

- *transakčná* replikácia – tento typ je dosť často používaný v praxi [20]. Typicky sa inicializuje použitím snapshot replikácie. Následne využíva logovacie súbory v zdrojovej databáze pre synchronizáciu s kópiami. Ide len o jednosmerný prenos zo zdrojovej databázy do kópií. V prípade zmien v zdrojovej databázy sa musia všetky zmeny reflektovať v kópii. Problém môže nastať v prípade externých zmien vykonaných v kópii, preto sa kópie zdrojovej databázy väčšinou používajú len pre čítanie.

- *peer-to-peer* replikácia – je to len transakčná replikácia so zopár vylepšeniami, ktoré umožňujú obojsmernú aktualizáciu dát medzi servermi. Každý uzol môže teda prijímať aj odosielať transakcie iným uzlom. Podrobnejšie je popísaná v [40].

- *merge* replikácia – umožňuje synchronizáciu dvoch a viac databáz. Taktiež začína so snapshot kópiou databázy. Tento prístup umožňuje aj off-line synchronizáciu. Teda kópie môžu byť odpojené istú dobu od originálu a synchronizujú sa až pri ďalšom spojení. Jednotlivé uzly môžu pracovať teda nezávisle a po určitej dobe spojiť svoje aktualizácie dát. Keďže aktualizácie môžu nastať na viacerých uzloch, môžu vzniknúť konflikty. Merge replikácia prináša niekoľko vstavaných spôsobov pre ich vyriešenie.

2.3.3 Vyvažovanie záťaže

Vyvažovanie záťaže databázového systému má za cieľ optimalizovať využitie dostupných zdrojov, maximalizovať priepustnosť, minimalizovať čas odozvy na požiadavky a vyhnúť sa preťaženiu jedného uzlu.

V prípade master-slave replikácie s len jedným hlavným uzlom sa aktualizácia dát nedá nijako rozdeliť, vždy prebieha priamo na hlavnej databáze a všetky ostatné si to musia zosynchronizovať. U master-master môže byť prvotná požiadavka na aktualizáciu na ľubovoľnom master uzle. Ostatné si to musia taktiež neskôr aktualizovať. Takže dochádza k aktualizácii a sieťovej komunikácii u všetkých uzlov pri aktualizácii dát.

Pri čítaní dát je ale situácia iná. Pre dosiahnutie uvedených vlastností je nutné rozumné vyváženie záťaže medzi jednotlivé uzly. Prístup k databáze zastrešuje služba, tzv. *load balancer*, ktorá vhodným algoritmom vyberá jeden zo serverov pre vybavenie požiadavky. Predpokladom je samozrejme fakt, že dáta sú na všetkých serveroch rovnaké. Niektoré z algoritmov sú [36]:

- *round robin* – najjednoduchší algoritmus, ktorý posiela požiadavky postupne na každý server. Výhodou je teda len jeho jednoduchosť, rýchlosť výberu serveru a predikovateľnosť ďalšieho výberu. Všetky servery teda dostanú rovnaké množstvo požiadaviek, čo môže byť vhodné v prípade, že majú dosť podobné parametre. Počet požiadaviek ale neodráža ich náročnosť a záťaž.

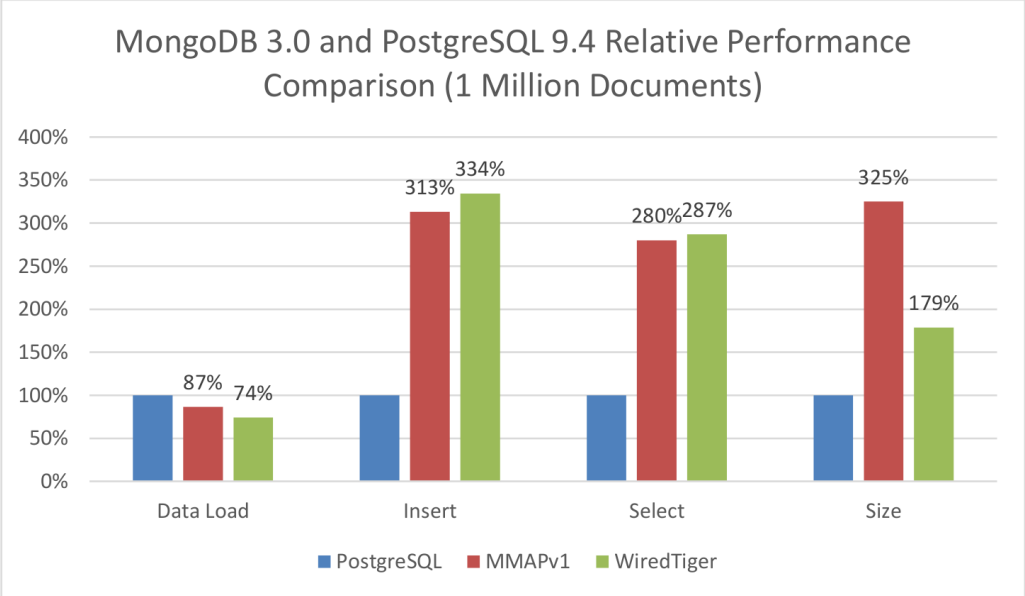
- váhovo rozdelená záťaž – jednoduchý algoritmus, ktorý rozdeľuje požiadavky na základe preddefinovaného pomeru. Jeho použitie je výhodné hlavne v prípade, že systém pozostáva zo serverov s výraznejšie rozdielnym výkonom.

- najmenej spojení – požiadavky sú v tomto prípade zasielané na server s najmenším počtom aktuálne aktívnych spojení. V situáciách, kedy majú požiadavky veľmi rozdielnu náročnosť môže tento prístup priniesť značnú výhodu. Vyžaduje si to ale aktívne monitorovanie požiadaviek.

2.4 Porovnanie výkonu SQL a NoSQL databáz

Výkon databázy – hlavne rýchlosť čítania a zápisu by mal byť len jedným z kritérií pri jej výbere. Kritérií a sledovaných charakteristík pre toto porovnávanie môže byť veľa.

Práca [19] sa zameriava na porovnanie populárneho SQL databázového systému PostgreSQL (testovaná verzia – 9.4.1) a taktiež populárneho NoSQL systému MongoDB (testovaná verzia – 3.0.2). Ide hlavne o porovnanie práce s JSON typmi pre reálne aplikácie. Práca využíva `pg_nosql_benchmark`[2]. Na obrázku 2.3 sú zobrazené výsledky. Vyššie hodnoty znamenajú viac času stráveného v databázových operáciách. Porovnáva PostgreSQL, MongoDB s MMAPv1 implementáciou ukladania dát a MongoDB s novšou WiredTiger architektúrou. Výsledky ukazujú dva až trikrát lepší výkon v prospech PostgreSQL.



Obr. 2.3: Porovnanie výkonu operácií v PostgreSQL, MongoDB s MMAPv1 architektúrou a WiredTiger architektúrou (prevzaté z [19]). Nižšia hodnota znamená lepší výkon.

Ďalšie porovnanie SQL a NoSQL databáz je napríklad v práci [25]. Ide konkrétne o SQL databázu PostgreSQL a NoSQL databázy Cassandra – stĺpcová DB, CouchDB – ukladá JSON dokumenty, MongoDB – dokumentová DB a RethingDB – JSON dokumenty.

[2] https://github.com/EnterpriseDB/pg_nosql_benchmark

Databázy boli nainštalované na zhluku štyroch serverov. NoSQL databázy majú natívnu podporu distribúcie, pre podporu distribúcie pre PostgreSQL bol použitý CITUS[3].

Testovanie bolo zamerané na zápis a čítanie. Konkrétne na dobu trvania operácií a priepustnosť. V prípade čítania a zápisu boli vykonané dva typy testov – sekvenčný prístup a náhodný. Pre sekvenčný prístup bol použitý jeden generátor dát. Pri náhodnom prístupe sa využili štyri generátory, každý obsahoval rovnaké dáta ale zapisoval/čítal určenú štvrtinu z nich v náhodnom poradí.

Doba trvania operácií bola meraná od začiatku prijatia operácie serverom až po jej kompletné dokončenie. Nezahŕňa teda čas sieťovej komunikácie, generovanie dát atď. Priepustnosť je pomer počtu vykonaných operácií ku času potrebnému pre ich vykonanie.

Najväčšiu priepustnosť dosiahla pri zápise databáza Cassandra a na druhom mieste skončil PostgreSQL. S rastúcim počtom uzlov $(1-4)$ narastala aj priepustnosť u všetkých databáz.

Pri testovaní čítania len na jednom uzle mali Cassandra, MongoDB aj PostgreSQL podobnú rýchlosť. Pri vysokom počte uzlov má ale najnižšiu dobu čítania MongoDB.

Grafy všetkých meraní sú uvedené v [25]. Výsledky sú nasledujúce:

- Cassandra – so zvyšujúcim počtom uzlov sa zvyšuje aj priepustnosť. Dosiahla najväčšiu priepustnosť pri zápise a taktiež pri čítaní. Celkovo vo väčšine testov je najlepšia. Nedosahuje najlepší výsledok pre dobu trvania čítania a zápisu, ale má najvyššiu priepustnosť (vďaka paralelizmu).
- MongoDB – používa len jeden master uzol, preto je doba trvania operácií aj priepustnosť stála aj pri zvyšujúcom sa počte uzlov. V priepustnosti zápisu má tretie miesto. Pri čítaní má takmer rovnaké výsledky ako prvá Cassandra.
- CouchDB – so zvyšujúcim počtom uzlov sa zvyšuje aj priepustnosť. V priepustnosti zápisu je najpomalšia.
- RethinkDB – používa len jeden master uzol, preto je doba trvania operácií aj priepustnosť stála aj pri zvyšujúcom sa počte uzlov. Vo väčšine testov je najpomalšia, prípadne mierne lepšia než CouchDB.
- PostgreSQL – používa len jeden master uzol, preto je doba trvania operácií aj priepustnosť stála aj pri zvyšujúcom sa počte uzlov. Pri čítaní má takmer rovnaké výsledky ako prvá Cassandra. Dosahuje najlepšie výsledky doby trvania zápisu a priepustnosti.

Medzi ďalšie zaujímavé porovnania SQL a NoSQL patrí napríklad [41], kde sa porovnávala MS SQL Express databáza s NoSQL databázami MongoDB, RavenDB, CouchDB, Couchbase, Cassandra a Hypertable. V testoch doby trvania čítania a zápsiu sa SQL databáza zaradila okolo tretieho až štvrtého miesta.

Práca [9] sa zameriava na porovnanie výkonu MongoDB a Oracle databáz. Testovanie obsahuje viacero testovacích sád a sleduje hlavne priepustnosť a rýchlosť operácii čítania, zápis, aktualizácii. MongoDB má prevahu vo všetkých testoch.

2.5 Distribuované systémy

Táto sekcia vychádza hlavne z [13].

[3] https://www.citusdata.com/

Distribuovaný systém pozostáva z množiny výpočtových jednotiek, ktoré sú prepojené sieťou. Výmena informácie medzi jednotkami prebieha sieťovou komunikáciou. Tieto systémy majú nasledovné vlastnosti:

- oneskorenie pri komunikácii je konečné, ale nepredvídateľné
- jednotky nemajú spoločný pamäťový priestor a komunikujú len zasielaním správ po sieti
- procesory nemajú spoločné fyzické hodiny
- zasielané správy nemusia byť doručené v poradí, v ktorom boli odoslané
- správy sa môžu stratiť, zahodiť, či duplikovať z dôvodu vypršania časového limitu, opätovného odoslania
- globálny stav distribuovaného výpočtu je tvorený stavmi procesov a komunikačných kanálov, kde stav procesu je daný stavom operačnej pamäte danej jednotky a stav kanálu je daný množinou správ v ňom
- systém môže byť reprezentovaný orientovaným grafom, kde vrcholy predstavujú výpočtové jednotky a hrany zobrazujú komunikačné kanály

Distribuované výpočty sú aplikované v mnohých oblastiach, napríklad:

- bezdrôtové mobilné siete
- sieť senzorov, napríklad v inteligentných systémoch
- paradigma publikovanie–odoberanie (angl. *publish-subscribe*), viď časť 2.5.1.
- distribuovaní agenti v agentných systémoch
- distribuované dolovanie znalostí z databáz

2.5.1 Distribuovaná komunikácia

V distribuovaných systémoch je veľmi populárnym spôsobom komunikácie zasielanie si správ medzi jednotlivými uzlami distribuovanej architektúry pomocou prostredníka nazývaného *broker*. Ide o asynchronnú komunikáciu s použitím paradigmatu publikovanie–odoberanie [6]. Hlavným cieľom je zjednodušiť vytvorenie decentralizovanej architektúry, zvýšiť odolnosť voči chybám a zaručiť vysokú dostupnosť [37].

Medzi najznámejšie protokoly patria Kafka a AMQP.

Kafka

Protokol Kafka bol vyvinutý spoločnosťou LinkedIn a zameraný na spracovávanie logov (on-line aj off-line). Hlavným cieľom je výkon pred spoľahlivosťou, má vysokú priepustnosť a nízke oneskorenie.

AMQP

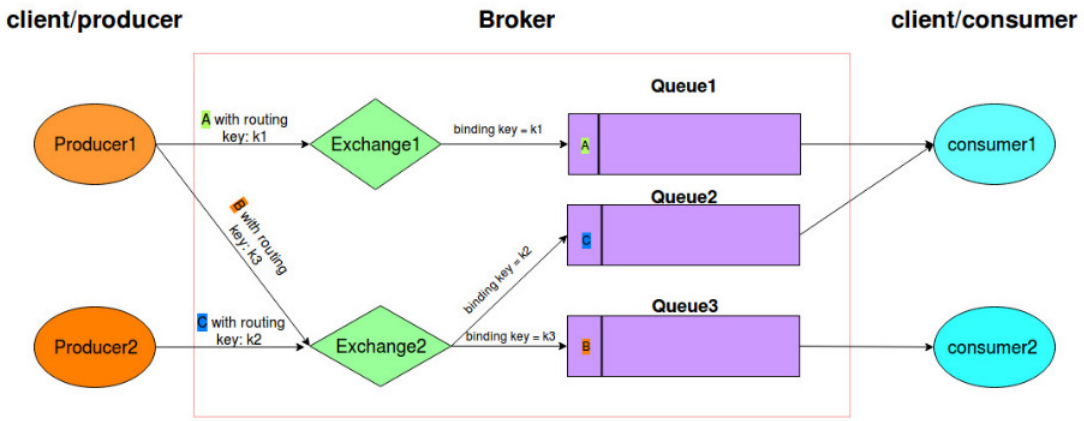
Ide o asynchronný protokol pre zasielanie správ. Pôvodne bol navrhnutý pre finančné a bankové systémy vyžadujúce spoľahlivosť a škálovateľnosť, čo malo značný vplyv na jeho vývoj [37].

Schéma tohto protokolu je zobrazená na obrázku 2.4. Správy od producenta nie sú zasielané priamo do front. Najprv sa dostanú do sprostredkovateľa (tzv. *exchange*) a odtiaľ

sú zaradené do front. Každá fronta sa prihlási na odoberanie určitých správ pomocou istého kľúča, tzv. *binding key*. Správa taktiež nesie svoj smerovací kľúč (tzv. *routing key*) a sprostredkovateľ teda posiela správy do front podľa zhody týchto kľúčov.

Konzument sa už len prihlási na odoberanie správ z požadovanej fronty. Nemusí sa už zaujímať o stav producenta a naopak, ani producent sa nemusí zaujímať o stav konzumenta.

Tento protokol je implementovaný napríklad v RabbitMQ[4], Apache Qpid [5], či JORAM [6].



Obr. 2.4: Schéma AMQP protokolu (prevzatá z [37]).

2.5.2 Celery

Nástroj Celery je flexibilný distribuovaný asynchronný systém pre spracovávanie enormného množstva správ, úloh. Je napísaný v jazyku Python, ale protokol môže byť implementovaný v ľubovoľnom jazyku [31].

Ide o takzvaný *task queue* mechanizmus pre distribúciu úloh medzi procesmi alebo aj počítačmi. Medzi klientom a procesmi vykonávajúcimi úlohy je typicky nejaký prostredník – tzv. *broker*. Jednotkou komunikácie je správa, ktorú zašle klient na daný broker. Správa obsahuje informácie o úlohe, ktorá sa má vykonať, jej parametre, prípadne ďalšie nastavenia, napríklad odložený čas spustenia správy, čas expirácie správy vo fronte a i. Broker ju zaradí do príslušnej fronty a dedikovaný proces v Celery odtiaľ odoberá tieto správy a prideľuje ich jednotlivým procesom na vykonanie. Výsledok spracovania úlohy môže byť požadovaný klientom. Na to sa dá využiť služba, kam bude výsledok uložený a klient si ho môže následne vyzdvihnúť. Celery poskytuje niekoľko takýchto vstavaných služieb (SQLAlchemy, Django, Redis, ...).

Celery systém môže obsahovať veľa procesov na vykonávanie úloh, na viacerých strojoch. Ďalšie jednotky je možné pridať za behu, napríklad pre zvýšenie výkonu, čím je zaistená vysoká dostupnosť a škálovateľnosť. Taktiež vie komunikovať s viacero broker službami.

[4] https://www.rabbitmq.com
[5] https://qpid.apache.org/
[6] https://joram.ow2.io/

Medzi populárne patrí RabbitMQ. Ďalšou možnosťou je využiť Redis[7], AmazonSQS[8], či Apache Zookeeper[9].

[7] https://redis.io
[8] https://aws.amazon.com/sqs/
[9] https://zookeeper.apache.org

Kapitola 3

Analýza škodlivého kódu

Antivírusové firmy denne dostávajú enormné množstvo vzoriek v rade tisícov a státisícov [12]. Manuálna analýza a tvorba detekčných pravidiel je pri takom počte nezvládnuteľná. Existuje veľa komerčných aj proprietárnych nástrojov pre urýchlenie a podporu tejto činnosti, až samotnú automatizáciu.

V obidvoch prístupoch sa využívajú dva základné princípy analýzy programov. Ide o statickú a dynamickú analýzu. Oba majú svoje výhody aj nevýhody a vzájomne sa môžu dopĺňať. Nasledujúci popis oboch prístupov vychádza z [18, 29].

Pri statickej analýze nie je nutné spustenie programu a nehrozí teda infikovanie systému. Navyše, v prípade napríklad skriptovacích jazykov je priamo k dispozícii zdrojový text. U binárnych programov bez možnosti nahliadnutia do zdrojového textu je proces statickej analýzy kódu zložitejší a využíva sa spätný preklad do jazyka symbolických inštrukcií (angl. *disassembling*), prípadne do vyššieho programovacieho jazyka (tzv. *dekompilácia*). V oboch prípadoch je ale problémom obfuskácia, kompresia (tzv. *packing*), či šifrovanie. Autori škodlivých programov (ale aj neškodlivých, napríklad z dôvodu ochrany interných znalostí) často využívajú rôzne techniky ukrytia pravej činnosti programu pred analytikmi a antivírusovými programami. Tento problém sa snaží riešiť dynamická analýza. Program je spustený vo virtuálnom stroji alebo emulátore a monitoruje sa jeho správanie (volanie systémových funkcií, komunikácia so vzdialeným serverom a pod.) [30].

3.1 Statická analýza

U binárnych programov sa typicky využíva spätný preklad do assembleru. Populárnymi nástrojmi sú napríklad IDA Pro[1], OllyDbg[2], objdump. Existujú už aj nástroje na preklad do vyššieho programovacieho jazyka. Preklad do jazyka `C` poskytuje napríklad nástroj RetDec[3] alebo Hex-Rays[4].

Pri preklade zdrojového súboru ale dochádza k zahodeniu niektorých informácií nepodstatných už pre činnosť programu a taktiež prekladače aplikujú rôzne optimalizácie. To prináša množstvo prekážok pre spätný preklad a výsledný zdrojový kód nebude identický s pôvodným kódom. Funkcionalita by mala byť zachovaná, ale čitateľnosť bude oveľa horšia. Na druhej strane, je to jednoduchšie na analýzu, než assembler.

[1] https://www.hex-rays.com/products/ida
[2] http://www.ollydbg.de
[3] https://www.retdec.com
[4] https://www.hex-rays.com/products/decompiler

3.2 Dynamická analýza

Dynamická analýza sa zameriava hlavne na správanie programu, ako je využívanie systémových funkcií, aplikačného rozhrania, sieťová komunikácia, práca so súbormi, ... Využíva sa pri tom napríklad odchytávanie volaní funkcií a sledujú sa ich parametre a návratové hodnoty.

V prípade dostupnosti zdrojového kódu ho môžeme jednoducho modifikovať a odchytávať funkcie priamo z kódu alebo využiť prekladač, napríklad v prekladači GCC s parametrom `-finstrument-functions` [5]. Tento scenár je ale pri analýze škodlivých programov takmer nereálny a využívajú sa napríklad techniky modifikovania binárneho kódu – upraviť sledované funkcie alebo inštrukcie volania funkcií [5]. Technika vkladania inštrukcií do zdrojového kódu a monitorovania rôznych aspektov programu sa nazýva inštrumentácia. Môže prebiehať na reálnom stroji, v sandboxe alebo aj emulovanom stroji [15].

Z dôvodu ochrany systému pred poškodením sa pri dynamickej analýze škodlivých programov využívajú práve tieto techniky [15]:

- sandboxing – ide o bezpečnostný mechanizmus, ktorý umožňuje spustenie programu v izolovanom prostredí s obmedzenými prostriedkami (operačná pamäť, deskriptory súborov, prístup na disk, ...)
- emulácia – je to virtuálne prostredie, ktoré môže simulovať ľubovoľnú architektúru (bez ohľadu na hostiteľský stroj). Kvôli nutnému prekladu inštrukcií medzi rozdielnymi architektúrami sa teda zvyšuje zložitosť a znižuje rýchlosť.
- virtualizácia – na rozdiel od emulácie musí byť inštrukčná sada kompatibilná s hostiteľským systémom [15]. Inštrukcie teda môžu byť spúšťané priamo na hostiteľskom stroji, čo prináša zrýchlenie. Výnimkou je privilegovaný mód, ktorý existuje iba virtuálne [5].

3.3 Delenie škodlivých programov

Na najvyššej úrovni môžeme programy rozdeľovať na škodlivé/neškodlivé/potenciálne škodlivé (tzv. PUP – *potentially unwanted program*). PUP programy sú na pomedzí a nie je presne definovaný a používaný rozdiel medzi škodlivým alebo neškodlivým programom. Viacero antivírusov sa v tomto rozchádza. Typicky sú niektoré prísnejšie a radia dané programy k škodlivým. Jedná z definícii pre PUP uvádza, že ide o program, ktorý je nainštalovaný spolu s iným vedome inštalovaným softvérom a licenčné podmienky naozaj aj informujú o inštalácii dodatočného, škodlivého programu. Problémom môže byť ale komplikovanejšia právnická formulácia podmienok, či ich úplné ignorovanie užívateľom [28].

U škodlivých programov, aj PUP, sa v praxi ešte zaviedlo jemnejšie delenie podľa spôsobu ich šírenia, vykonanej škody, atď. na typy. Niekedy ale nie je možné priradiť programu len jeden typ. Ďalej sa zaužívalo delenie do rodín na základe spoločných charakteristík a prípadne autora.

Typy programov

Pri určovaní typu programu sa vychádza hlavne zo správania programu. Aj tu môžeme ale naraziť na priradenie odlišných typov jednému programu viacerými anitvírusmi. Dôvodom môže byť hlavne to, že program vykazuje správanie viacerých typov. Medzi základné typy patria:

- adware – program šíriaci reklamu. Často ale môže aj zhromažďovať osobné údaje, históriu prehliadania, a i. Niekedy je radený aj do kategórie potenciálne škodlivých programov.
- backdoor – ide o metódu tajného obchádzania bežnej autentifikácie a zabezpečenie si vzdialeného prístupu do systému.
- banker – je to typ malvéru, ktorý sa pokúša ukradnúť prihlasovacie údaje k internetovému bankovníctvu.
- bootkit – program, ktorý využíva Master Boot Record (MBR) na disku, čo mu umožňuje spustiť sa spolu s operačným systémom a nie je možné ho detekovať štandardnými prostriedkami operačného systému, pretože MBR je mimo súborového systému.
- crypter – program, ktorý obfuskuje, šifruje, maskuje malvér za účelom sťaženia jeho detekovania ochrannými programami. Takto upravený program vyzerá byť úplne neškodný, pokým sa nenainštaluje, prípadne nespustí.
- dropper – ide o typ trojan malvéru, ktorého úlohou je nainštalovať iný malvér. Buď už sám obsahuje iný škodlivý kód, alebo ho stiahne cez internet.
- exploit – program, ktorý sa snaží zneužiť zraniteľnosti systému a umožňuje útočníkovi vzdialený prístup do systému.
- fileinfector – pridáva škodlivý kód do iných spustiteľných súborov a mení tak jeho správanie. Tým ale sťažuje jeho odstraňovanie, pretože väčšina užívateľov nechce prísť o svoje programy.
- injector – ide o malvér, ktorý vkladá sám seba do iných procesov alebo súborov, čím sa skryje pred užívateľom.
- keylogger – program monitorujúci každé stlačenie kláves. Dáta ukladá potom lokálne a útočník si ich neskôr vyzdvihne alebo ich rovno odosiela cez internet.
- phishing – technika zasielania e-mailov, ktoré sa snažia vyzerať vierohodne, ale cieľom je ukradnúť dôverné dáta užívateľa.
- pws (password-stealer) – program, ktorý na pozadí zbiera informácie o systéme, užívateľoch a snaží sa ukradnúť prihlasovacie údaje, užívateľské kontá, dôveryhodné informácie.
- ransomware – zabraňuje užívateľovi dostať sa k osobným, či systémovým súborom a požaduje výkupné za ich sprístupnenie. Typicky ide o zašifrovanie týchto súborov.
- rat (remote access trojan) – program, ktorý umožňuje vzdialenú kontrolu cieľového systému.
- screenlocker – druhý typ ransomware programu, ktorý ale namiesto šifrovania súborov zabraňuje užívateľovi v používaní systému.
- toolbar – tento typ patrí skôr do kategórie PUP a typicky ide o infikovanie prehliadača grafickými prvkami – vyhľadávacie polia, tlačítka, menu, propagácia reklamy a i.

- trojan – program vydávajúci sa za legitímny softvér, ale v skutočnosti vykonáva nejakú škodlivú činnosť z vyššie uvedených programov.
- trojan-sms – trojan špecifický pre mobilné telefóny, ktorý odosiela textové správy na spoplatnené čísla.
- virus – program, ktorý sa sám kopíruje a vkladá do iných programov či súborov.
- worm – typ malvéru, ktorý sa sám kopíruje a šíri po sieti s využitím zraniteľností.

Samozrejme typov môže byť ešte viac, záleží to už asi na konkrétnej definícii medzi analytikmi, či v rámci antivírusovej spoločnosti.

Konvencie pomenovania škodlivých programov

Táto časť vychádza z [27].

Ďalším delením škodlivých programov je príslušnosť k nejakej rodine. Ide o skupinu vzoriek s podobnými charakteristikami, prípadne rovnakým autorom. Názvy rodín sú často tvorené antivírusovými firmami alebo inými analytikmi. Niekedy ide o mená odvodené od textu, ktorý autor zakomponoval do programu, či sám autor použil nejaký vlastný identifikátor. Napríklad v prípade ransomware rodiny Locky dochádza k zašifrovaniu súborov a pridaní prípony *locky* k názvu súboru. Pre tvorbu názvov rodín nie je zavedený žiaden štandard a teda niekedy vznikajú aj k viacerým pomenovaniam jedného programu.

Uvedené rozdelenie na druh škodlivosti, konkrétnejší typ a rodinu sa používa pre vytváranie pomenovania detekčných definícii/pravidiel. Teda pravidlo, ktorým vieme detekovať daný škodlivý súbor zahŕňa väčšinou všetky tieto informácie. Syntax pomenovania týchto detekcií sa ale odlišuje. Názov od jedného antivírusu `Trojan/Win32.Bladabindi` môže vyzerať aj takto: `Backdoor.MSIL.Bladabindi.a (v)`, `BKDR_BLADABI.SMC`. Všetky uvedené názvy sa zhodujú v rodine (Bladabindi), ale v prvom prípade je uvedený typ trojan, a posledný príklad používa nejakú vlastnú skrátenú verziu pre backdoor.

O zavedenie jednoty do názvov detekcií sa snažila skupina CARO konvenciou v roku 1991 [1]. Okrem formátu názvov detekcií navyše uvádza pravidlá a obecné odporúčania pre vytváranie nových pomenovaní pre rodiny. Navrhovaný formát detekcie je takýto: `Family_Name.Group_Name.Major_Variant.Minor_Variant[:Modifier]`.

Iný prístup zvolila iniciatíva CME [14], ktorá sa snažila zaviesť jedinečný identifikátor v tvare CME-N, kde N je prirodzené číslo.

3.4 Clusty

Táto sekcia vychádza z [16].

Jedným z nástrojov pre automatické spracovanie prichádzajúcich vzoriek v Avast Software je nástroj Clusty. Jeho cieľom je zhluková analýza prichádzajúcich programov (tzv. vzorky). Ide teda o proces rozdeľovania objektov do tried (zhlukov) na základe ich podobnosti. Objekty v rámci jednej triedy sú si veľmi podobné a zároveň nie sú príliš podobné objektom iných tried. Pre určenie podobnosti je často nutné vybrať vhodné atribúty a taktiež aj vzdialenostné funkcie. Objekty v rámci jednej triedy môžeme potom spracovávať hromadne.

3.4.1 Zhluková analýza

V nástroji Clusty sa na najvyššej úrovni rozdeľujú vzorky do kategórii podľa platformy – PE, .NET, Mach-O, APK, ... Vzorky sa medzi týmito kategóriami nijako nemiešajú. Ide o samostatné zhluky. Pre každú z nich je implementovaná analýza pre získanie atribútov. Je to prevažne statická analýza pomocou voľne dostupných, či interných nástrojov. Pre niektoré kategórie sú k dispozícii aj informácie z dynamickej analýzy. Na základe heuristicky zvolených atribútov sa vzorky rozdeľujú do zhlukov. Každý zhluk by mal obsahovať vzorky len z jednej rodiny. Mix viacerých rodín v jednom zhluku je veľmi nežiadúci. Naopak, rozdelenie jednej rodiny do viacerých zhlukov je tolerované.

3.4.2 Klasifikácia zhlukov

Ďalšou úlohou tohto nástroja je klasifikácia zhlukov. Je to určenie triedy škodlivosti (škodlivý – *malware*, potenciálne škodlivý – *PUP*) alebo je to zhluk neškodných vzoriek (tzv. *clean*). Ďalej sa určuje typ programu – trojan, ransomware, worm, . . . a treťou časťou klasifikácie je rodina.

Vytvorené zhluky sú dostupné pomocou webového rozhrania analytikom, ktorí môžu takto manuálne klasifikovať. Preto je dôležité, aby bol jeden zhluk tvorený len vzorkami jednej rodiny. Analytik tak nemusí prechádzať všetky vzorky, ale klasifikuje celý zhluk naraz. Vzorky, ktoré budú pridané do tohto zhluku neskôr sú automaticky označené tou istou klasifikáciou.

Keďže aj zhlukov je stále veľmi vysoký počet, tak bola implementovaná aj ich automatická klasifikácia v práci [27]. Využívajú sa dva zdroje informácií – detekcie od antivírusov a YARA pravidlá. Oba prístupy sú popísané v ďalšej časti.

Klasifikácia pomocou antivírusových detekcií

Názvy detekcií od iných antivírusov sú k dispozícii pre každú vzorku zhluku a pochádzajú z webovej služby VirusTotal[5]. Z týchto detekcií sa dá často extrahovať meno rodiny a typ. Občas je k dispozícii aj trieda škodlivosti, ale táto informácia sa dá už odvodiť od typu a vzhľadom na to, že je to detekcia pre škodlivý program, tak určite nepôjde o triedu *clean*. Niekedy ale je detekcia veľmi generická, pretože antivírus nevie presne určiť, o čo ide. Takáto detekcia bola pravdepodobne vygenerovaná automaticky a obsahuje len triedu škodlivosti (napr. `Win32:MalwareGen`). Niekedy sa ako názov rodiny používa označenie zneužitej zraniteľnosti podľa CVE[6].

Pre všetky tri typy informácií (trieda, typ, rodina) platí, že rôzne antivírusy používajú trocha iné varianty alebo skratky. Táto prekážka sa obchádza len statickým mapovaním na kanonické hodnoty použité v nástroji Clusty.

Tento automatický klasifikátor teda analyzuje jednotlivé názvy detekcií a snaží sa z nich automaticky určiť triedu, typ, rodinu. V prípade triedy a typu je to jednoduchšie, pretože má k dispozícii zoznam možných hodnôt. Určenie rodiny je už ťažšie, pretože detekcie môžu obsahovať generické názvy alebo viacero reťazcov, ktoré by mohli byť rodinou. Tento sémantický problém je sčasti riešený zoznamom generických reťazcov, ktorý sa ale musí priebežne aktualizovať. V prípade niektorých antivírusov je syntax ich detekcií tak presná, že umožňuje jednoznačné určenie rodiny. Podrobnejšie je tento problém popísaný v [27]. Z extrahovaných informácií sa na základe počtu výskytov jednotlivých položiek a ich generičnosti,

[5] https://www.virustotal.com
[6] https://cve.mitre.org/

prípadne priority vyberú najlepšie hodnoty pre klasifikáciu. Nevýhodou pri určovaní rodiny je malá znalosť aliasov, teda zoznam rôznych názvov označujúcich v skutočnosti tú istú rodinu.

Klasifikácia pomocou YARA pravidiel

Druhým zdrojom pre automatické klasifikovanie vytvorených zhlukov sú YARA pravidlá[7]. Je to nástroj používaný na identifikovanie a klasifikovanie škodlivých programov, ale umožňuje vytvárať popisy ľubovoľných súborov.

Pravidlá obsahujú [38]:

- metainformácie – autorom definované kľúče a ich hodnoty, typicky sa používajú pre popis rodiny, typu, škodlivosti detekovaných súborov
- reťazce – hexadecimálne, textové, regulárne výrazy
- podmienky – booleovské, relačné, aritmetické a bitové operátory s použitím definovaných reťazcov definujú podmienky detekovania súboru daným pravidlom

Ak všetky vzorky daného zhluku boli detekované daným YARA pravidlom, tak sa jeho metainfomácie môžu použiť priamo pre vytvorenie klasifikácie. V ukážke pravidla 3.1 ide o pravidlo pre detekovanie ransomware programov na základe výskytu istej postupnosti znakov na počiatočnej pozícii súboru.

```
rule teslacrypt_known_sequences
{
  meta:
    author = "Name Surname"
    description = "detection based on known sequences"
    strain = "MalwareFamily"
    version = "campaign_A"
    type = "ransomware"
    severity = "malware"
  strings:
    $s00 = "42DAi(}{>!DA":AD{:AD!@#)(!-=[{}ASDOAS!@#GDO^I
  condition:
    $s00 at 0x0
}
```

Kód 3.1: Ukážka YARA pravidla detekujúceho malvér MalwareFamily na základe výskytu reťazca v súbore na pozícii `0x0`.

3.5 Ďalšie nástroje

Medzi ďalšie nástroje, ktorých výstup môže byť zaujímavý pre nástroj vytvorený v tejto práci patrí Cuckoo sandbox, emulátor PE súborov a statický YARA skener. Sandbox a emulátor vykonávajú dynamickú analýzu súborov. Nad výsledkami ich analýz sú opäť testované

[7] https://virustotal.github.io/yara/

YARA pravidlá a v prípade chytenia sa pravidla je taktiež táto informácia zaslaná do RabbitMQ. YARA skener len skenuje súbory sadou YARA pravidiel a zasiela zoznam chytených pravidiel na RabbitMQ server.

Kapitola 4

Návrh platformy

V tejto kapitole je navrhnutá vysokovýkonná platforma pre účely výskumu malvéru. Pre jednoduchosť je ďalej táto platforma pre zhromažďovanie a vyhodnocovanie tagov označovaná len ako Tagger.

V tejto práci je popísané hlavne ukladanie a reprezentácia tagov, súborov a detekčných definícií. Rozšírenie o IoC (indikátory škodlivej činnosti) a prípadne iné objekty by bolo tejto reprezentácii veľmi podobné. Pre všeobecnosť sa ďalej niekde vyskytuje len pojem objekt. Nezávisí teda na tom, či ide o súbor, detekciu alebo IoC.

Vytvorenie tejto platformy pre podporu výskumu malvéru môžeme rozdeliť na niekoľko častí. Jej základom je databáza na ukladanie tagov pre súbory, detekčné definície, IoC atď. Informácie o objektoch a ich potenciálnych tagoch je potrebné získavať z nástrojov popísaných v kapitole 3. Hlavným zdrojom informácií je nástroj Clusty, ktorý ich poskytuje asi najväčšie množstvo. Vzhľadom na jeho prebiehajúci vývoj ešte neposkytuje funkcionalitu, akú si Tagger vyžaduje, a preto je v rámci tejto práce navrhnuté a implementované aj rozšírenie tohto nástroja. Pôjde o vylepšenie automatického klasifikovania (tagovania) v nástroji Clusty, či vylepšenie a rozšírenie rozhrania pre prístup k jeho informáciám.

Získané informácie musí Tagger vyhodnocovať a porovnávať s už existujúcimi dátami uloženými v databáze, pretože pre jeden súbor, či definíciu môže vyvíjaná platforma dostať viacero potenciálne rozličných informácií.

Pre poskytnutie zhromaždených informácií je potrebné vytvoriť rozhranie na komunikáciu s touto platformou. Pôjde určite o aplikačné programové rozhranie (API) a webové rozhranie pre jednoduchosť používania. Webové rozhranie bude použité taktiež pre prezentáciu generovaných štatistík šírenia malvéru a pod. Tieto štatistiky budú vytvorené na základe dát uložených v Taggeri a pomocou dát uložených v databáze spoločnosti Avast Software o aktivite škodlivých súborov u koncových užívateľov.

4.1 Databáza

V kapitole 2 sú popísané databázové systémy a zameriava sa hlavne na ich výkon, škálovateľnosť a replikáciu. Pri výbere databázy pre Tagger je nutné zvážiť viacero aspektov. Dôležité vlastnosti tejto platformy, ktoré ovplyvňujú výber databázy sú:

- Vysoký výkon – je dôležitý kvôli enormnému množstvu denne spracovávaných, analyzovaných a klasifikovaných súborov a spoločnosti Avast. Teda ide o rýchlosť zápisu nových dát. Dôležité je ale aj čítanie dát. Dôvody sú hneď dva. Čítanie dát nutné pre aktualizáciu objektu – rozhodnutie o aktualizovaní tagu pre objekt bude prebiehať

na aplikačnej úrovni. Ďalším konzumentom dát z databázy budú užívatelia, analytici a samotný Tagger pri generovaní štatistík.

Pravdepodobne bude nutný vyšší výkon pre čítanie, než zápis. Pri zápise môžeme počítať so státisícami objektov denne. Čítanie môže byť oveľa intenzívnejšie. V tomto prípade môže výrazne výkon zvýšiť replikácia dát, čo nám samozrejme zaistí aj zvýšenie dostupnosti dát.

- Škálovateľnosť – dôležitý faktor z dôvodu predpokladaného nárastu nových dát v budúcnosti.
- Konzistencia dát – Tagger bude vyžadovať zabezpečenie plnej konzistencie dát. Dôvodom sú najmä nebezpečné dôsledky pri nesprávne zapísanom tagu pre objekt. Používatelia môžu na základe toho nesprávne vygenerovať detekciu, ktorá sa dostane ku koncovým užívateľom spoločnosti Avast, iné nástroje môžu takto prehliadnuť škodlivý alebo čistý objekt a pod. Taktiež generované štatistiky môžu byť skreslené vplyvom nesprávnych tagov.
- Komplexnosť dotazov – okrem jednoduchého získavania tagov pre objekty musí Tagger podporovať aj zložitejšie analytické dotazy.
- Veľkosť dát – kvôli veľkému objemu dát je vhodnejšie ukladať dáta v normalizovanej forme. Každý objekt má priradený tag a veľké množstvo objektov má ten tag rovnaký. Pre spomalenie nárastu databázy je vhodnejšie normalizovať dáta.

Pri daných požiadavkách môžeme na najvyššej úrovni rozhodovať o tom, či použijeme SQL alebo NoSQL databázu.

- Z ich porovnania v sekcii 2.4 vychádza lepšie NoSQL databáza (hlavne Cassandra), čo sa týka škálovateľnosti a teda rastúceho výkonu.
- Nevýhodou NoSQL databáz pre Tagger je hlavne problém s konzistenciou dát. Chyba v jednom tagu pre jednu detekciu, ktorá za určitý časový úsek zablokovala u koncových užívateľov veľké množstvo škodlivých súborov môže výrazne skresliť vygenerované štatistiky a trendy šírenia malvéru.
- Ďalšou výraznou nevýhodou je redundancia veľkého množstva dát pri denormalizovanej forme uložených dát.
- Dáta ukladané v Taggeri budú štruktúrované a nie je problém pre ne vytvoriť jasnú normalizovanú schému. Preto nie je problém zvoliť relačnú SQL databázu.
- V porovnaní výkonnosti v sekcii 2.4 sa najčastejšie z relačných databáz objavovala PostgreSQL a s veľmi dobrými výsledkami. Dosť dobre konkurovala databáze Cassandra v rýchlosti (samozrejme pre nižšom počte uzlov).
- Nevýhodou SQL databázy môže byť vertikálne škálovanie.
- K jasným výhodám SQL databázy ale ďalej patrí podpora replikácie, má robustnú funkcionalitu (viď sekciu 2.1) a je rozširovateľná.
- Z praktických výhod použitia SQL databázy je tu ešte výhoda možnosti spravovania databázy IT tímom v spoločnosti Avast Software.

Z uvedených dôvodov teda bola zvolená pre Tagger platformu databáza PostgreSQL.

4.1.1 Ukladanie dát

Dáta uložené v Taggeri sa budú primárne týkať tagov. Okrem nich ale bude nutné ukladať aj výsledky štatistických analýz, ale to nemá súvislosť s tagmi a nevyžaduje si to vysokovýkonnú databázu. Dáta môžu byť prípadne neštruktúrované a uložené aj v NoSQL databáze pre jednoduchosť. Tejto problematike sa ale bude venovať časť o generovaní štatistík. Táto sekcia je zameraná na ukladanie tagov.

Tag je teda n-tica – severita (malvér, PUP, tool, čistý objekt, ...), typ danej severity (červ, trojan, bot, ...), rodina (len pre škodlivé objekty), varianta danej rodiny, cieľová platforma.

Objekty môžu byť tagované viacerými nástrojmi a tagy sa môžu líšiť. Tagger preto musí vyberať najlepší tag zo všetkých poskytnutých informácií. Z tohto dôvodu bude ďalej dobré ukladať si informáciu o zdroji, teda službe, ktorá daný tag vytvorila. Či už ide o klasifikáciu z nástroja Clusty, YARA skeneru alebo manuálnu klasifikáciu analytikom.

Niektoré služby môžu samozrejme po istom čase zmeniť klasifikáciu objektu, a preto si Tagger bude ukladať aj časové razítko ku každému tagu. V prípade nesprávneho poradia príchodu takýchto aktualizovaných tagov si Tagger s nimi poradí (predpokladom je samozrejme časové razítko ako súčasť tagu zaslaného danou službou).

Ďalšou dôležitou informáciou pre tag je miera jeho dôveryhodnosti. Každý nástroj využíva iné informácie a algoritmy pre vytváranie tagov. Niektoré sú spoľahlivejšie než iné. Napríklad manuálna klasifikácia analytikom v nástroji Clusty, či YARA pravidlo taktiež vytvorené analytikom by mali byť spoľahlivejšie než automatická klasifikácia v nástroji Clusty postavená na antivírusových detekciách. Táto automatická klasifikácia ma taktiež rozličnú spoľahlivosť, keďže pracuje s variabilným počtom dostupných detekcií.

Schéma tabuliek potrebných pre reprezentáciu tagov pre súbory je na obrázku 4.1. Základné informácie uchovávané o súbore sú jeho jedinečný identifikátor, tag samotný a zdroj tejto informácie:

- Ako jedinečný identifikátor súborov je použitý SHA-256 hash. Dôvodom je hlavne jeho použitie vo väčšine nástrojov použitých ako zdroj informácií, dostatočná unikátnosť, či aj obecné použitie medzi inými antivírusovými nástrojmi. Toto je jediná informácia týkajúca sa súboru samotného, ktorá sa ukladá. Metadáta, názov súboru a pod. by boli len redundantné informácie, ktoré sa dajú podľa hashu vyhľadať v iných nástrojoch, či on-line.

- Tag predstavuje už spomenutú päticu informácií – trieda škodlivosti (angl. *severity*), typ malvéru a PUP, rodina, varianta a platforma. Najpodstatnejšou položkou je trieda škodlivosti. Bez tejto informácie sú ostatné bezvýznamné.

- Zdroj daného tagu sa ukladá hlavne z dôvodu spätného vyhľadania jeho autora a taktiež pre rozhodovanie sa pri porovnávaní rozličných tagov. Vo väčšine prípadov ide o uloženie názvu služby prípadne jednej podrobnejšej informácie – názvu YARA pravidla u YARA skenerov, typ automatickej klasifikácie z nástroja Clusty a pod.

Ďalšie podporné informácie budú:

- Časové razítko vytvorenia daného tagu v danom zdroji. V prípade, že zdroj neposkytuje túto informáciu, použije sa čas, kedy Tagger obdržal informáciu o tomto tagu. Táto informácia sa môže využiť neskôr ako jedna z podmienok vyhodnocovania prioritnejšieho tagu. Aj keď váha môže byť veľmi malá a pôjde asi len o poslednú podmienku

rozhodnutia za účelom uchovávania novšieho tagu. V prípade paralelného spracovávania prichádzajúcich správ sa môže novšia správa spracovať skôr a tag zo staršej správy by mohol prepísať následne tag z novšej.

- Miera dôveryhodnosti daného tagu. Niektoré nástroje poskytujú túto hodnotu sami, u iných si túto hodnotu musí Tagger vytvoriť sám. Táto hodnota sa môže taktiež využiť pri prioritizovaní tagov.

- História predchádzajúcich tagov pre daný objekt, keďže Tagger počíta len s jedným najlepším tagom pre objekt. Obsahuje konkrétny tag, jeho zdroj a časové razítko. História je zavedená hlavne z dôvodu jednoduchšieho hľadania chýb a ladenia. Ďalší dôvod je to, že Tagger sa vždy pokúsi určiť pre daný objekt jeden najlepší tag z aktuálne dostupných informácií. Druhou možnosťou je uchovávanie informácií o všetkých dostupných informáciách, ale to by znamenalo oveľa vyššiu náročnosť na kapacitu databázy. V praxi analytika zaujíma väčšinou aj tak len jedna informácia o objekte, takže na určitej úrovni musí stále dochádzať k výberu najlepšieho tagu.

V rámci tejto práce je pridaná podpora pre tagovanie súborov a detekčných definícií. Pridávanie ďalších objektov je už analogické.

Každý typ objektu, teda súbory a detekčné definície má vlastnú tabuľku. Dôvodom je mierne odlišný prístup k reprezentácii objektu. Zatiaľ čo u súborov postačuje SHA-256 hash, u detekčných definícií si potrebujeme uložiť detailnejšie informácie z dôvodov zjednodušenia analytických dotazov.

Detekčné definície majú svoj unikátny názov a taktiež informáciu o svojom pôvode. Teda väčšinou ide o označenie algoritmu, ktorý ich vytvoril, alebo je to typ definície (napr. detekcia založená na výskyte reťazca v súbore). Táto informácia nie je identifikovateľná nejakým jednoduchým spôsobom z názvu detekcie a získava sa pomocou interného nástroja v spoločnosti Avast. Následne sa do databázy uložia obe informácie – názov a typ detekcie.

Typ sa môže využiť napríklad pri analýze typu škodlivého programu najčastejšie blokovaného danou detekciou, či generovanie štatistík úspešného blokovania súborov u koncových užívateľov daným typom detekcií.

Schéma tabuliek pre detekčné definície je teda rovnaká ako schéma pre súbory 4.1. Jediný rozdiel je, že namiesto tabuliek `sample_sha256_tag` a `sample_sha256_tag_history` sa použijú tabuľky `avast_detection_definition_tag` a `avast_detection_definition_tag_history`. Prvá z nich nebude obsahovať `sha256` stĺpec, ale dva stĺpce – `detection_name` a `type`.

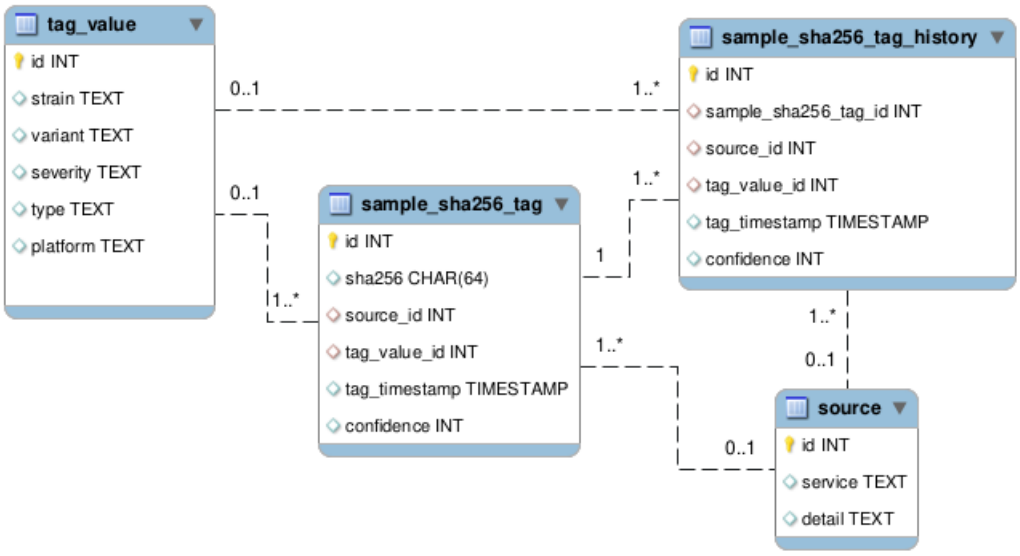
U ďalších objektov potenciálne ukladaných v Taggeri (IoC, URL, . . .) sa môžu vyskytnúť podobné odlišnosti pri ich reprezentácii v databáze a z tohto dôvodu je lepšie ukladať ich vo vlastných tabuľkách.

Tabuľky `tag_value` a `source` sú spoločné pre všetky objekty.

4.1.2 Výkon a škálovateľnosť

Možnou nevýhodou zvolenej databázy PostgreSQL je vertikálna škálovateľnosť. Vyžaduje si výkonnejší a drahší hardvér. Oproti horizontálnej škálovateľnosti nemôže bežať na viacerých strojoch.

Čo sa týka škálovateľnosti čítania dát, PostgreSQL podporuje replikáciu. V tomto prípade môžeme využiť viacero serverov, ktoré budú replikovať dáta z primárnej databázy. Zaistí sa takto vyššia priepustnosť prístupu k dátam a vyššia odolnosť voči výpadkom.



Obr. 4.1: Schéma tabuliek na reprezentáciu tagov pre súbory v nástroji Tagger.

Okrem vertikálneho škálovania databázy a replikácie by sme mohli využiť aj rozdelenie dát (nejde priamo o sharding ako je popísaný v sekcii 2.3.1). Môžeme vytvoriť vlastnú databázu na vlastnom serveri pre každý typ objektu. Teda v jednej databáze budú uložené tagy pre súbory, v ďalšej databáze, na inom serveri, budú tagy pre detekčné definície atď. Medzi týmito objektmi nemá Tagger žiadne súvislosti ani prepojenia. Sú to samostatné jednotky. V prípade rozdelenia do samostatných databáz by sme museli v každej z nich mať vlastné tabuľky pre zdroje a tagy samotné (tag ako n-tica informácií – severita, typ, rodina, . . .). Zdrojov informácií je ale veľmi málo a celkový počet rôznych tagov je len v tisícoch. Takáto redundancia je teda prípustná. Cenou za to bude opäť správa väčšieho počtu databáz a náročnejšia implementácia pre prístup k osobitný prístup k jednotlivým objektom. Táto možnosť zvýšenia výkonu ale nie je implementovaná v rámci tejto práce.

4.2 Zbieranie informácií

Nástroj Tagger neprodukuje žiadne tagy sám. Ide len o agregáciu a vyhodnocovanie informácií vytvorených v rozličných nástrojoch spoločnosti Avast.

Súbory prichádzajúce na analýzu sa postupne dostávajú do jednotlivých nástrojov. Ich čas príchodu a doba analýzy sa môžu výrazne líšiť. Všetky nástroje po skončení analýzy, vytvorení tagu, či inej dôležitej udalosti, odosielajú správy na RabbitMQ server, kde si ich Tagger môže odchytávať. Každá nová informácia sa musí v nástroji Tagger vyhodnotiť. Sú dva možné scenáre:

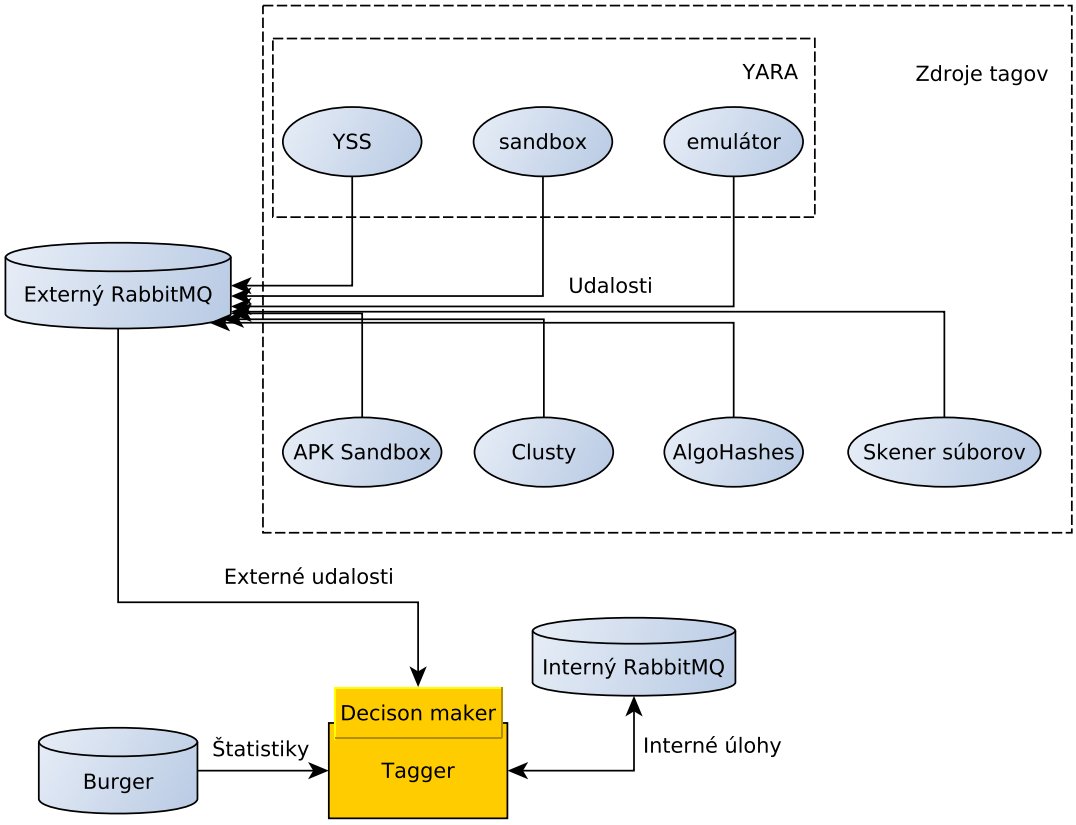
- Objekt sa v databáze ešte nenachádza – bude vložený ako nový záznam. Samotný tag objektu sa v databáze už môže nachádzať (tabuľka `tag_value`). Zdroj už taktiež môže byť v databáze (tabuľka `source`). V prípade, že tam ešte nie sú, budú vložené. Objekt sa teda vloží do konkrétnej tabuľky (`sample_sha256_tag` pre súbory atď.) so správnymi cudzími kľúčmi odkazujúcimi na tag a zdroj. Do tabuľky objektu sa vyplnia

všetky potrebné hodnoty – jeho identifikátor, miera dôveryhodnosti k aktuálnemu tagu a časové razítko aktuálneho tagu.

- Objekt je už v databáze – dôjde k načítaniu aktuálneho tagu z databázy, porovnanie s novozískanou informáciou, výber vierohodnejšej informácie podľa priorít popísaných v časti 4.3 a prípadná aktualizácia tagu s presunom pôvodného tagu do histórie.

  Pri aktualizácii tagu dôjde len k zmene údajov v tabuľke objektu. Upravia sa cudzie kľúče odkazujúce na tag a zdroj tagu, upraví sa miera dôveryhodnosti k tagu a jeho časové razítko. Samozrejme, ak sa nový tag a zdroj nenachádzajú vo svojich tabuľkách, tak budú tam vložené, rovnako ako v predchádzajúcom prípade pri vkladaní nového objektu.

Napojenie nástroja Tagger na všetky služby poskytujúce informácie o tagoch je na diagrame 4.2. Jednotlivé služby sú popísané v ďalšej časti.



Obr. 4.2: Diagram prepojenie Tagger-u s ostatnými službami

Clusty

Zo správ odosielaných nástrojom Clusty sú pre Tagger užitočné tieto:

- Zmena klasifikácie zhluku vzoriek – Klasifikácie zhlukov sa aktualizujú časom, pretože do zhluku pribúdajú nové vzorky, nové informácie o detekciách nad týmito vzorkami

atď. V prípade tejto udalosti dostane Tagger k dispozícii identifikátor daného zhluku, jeho základný popis a novovytvorenú klasifikáciu, teda tag. Pomocou API je teda potrebné získať zoznam všetkých súborov nachádzajúcich sa v danom zhluku a zoznam detekčných definícií týchto vzoriek. Nový tag sa aktualizuje pre súbory aj detekcie, tak ako je uvedené vyššie. Teda pre každú jednu vzorku a detekciu sa vyhodnotí, či bude vložená, aktualizovaná alebo bez zmeny. Vzhľadom na to, že ide o väčšie množstvo objektov naraz, tak je výhodnejšie pracovať so všetkými spolu. Teda pri zápise a čítaní z databázy využiť hromadné operácie, nie jednotlivo pre každý objekt.

- Pridanie súboru do zhluku – pre súbor, ktorý bol pridaný do nejakého zhluku budeme automaticky uvažovať, že jeho novým tagom je tag toho zhluku. S touto hodnotou ho aj aktualizujeme, či vložíme ako nový.

- Aktualizácia zoznamu detekčných definícií vzoriek v rámci zhluku – v tomto prípade ide len o pridanie nových detekčných detekcií ku vzorkám daného zhluku. Dané detekcie sa teda aktualizujú v Tagger databáze s použitím tagu tohto zhluku.

YARA

Medzi ďalšie nástroje, z ktorých Tagger môže čerpať, patrí Cuckoo sandbox, emulátor a statický YARA skener. Ich informácie sa použijú len pre tagovanie súborov. Prvé dva nástroje poskytujú kompletnú dynamickú analýzu, ale z toho sa použije len informácia o zasiahnutých YARA pravidlách na výsledkoch dynamického správania. YARA skener poskytuje informácie o zasiahnutých YARA pravidlách len na základe statickej analýzy.

Každé YARA pravidlo obsahuje informáciu o severite, type, rodine a variante. Chýba informácia o platforme. Taktiež chýba aj miera dôveryhodnosti tagu. YARA pravidlá majú ale vlastný atribút označujúci ich spoľahlivosť a ten môžeme použiť pre odvodenie hodnoty spoľahlivosti ako to vyžaduje Tagger.

Z týchto informácií sa teda vytvorí tag a použije na vloženie do databázy či aktualizáciu tagu daného súboru.

APK sandbox

Primárnym nástrojom pre informácie o APK (Android aplikácie) súboroch je ďalší sandbox, ktorý poskytuje pre samotné súbory informácie o ich typoch, prípadne aj rodinách. Ide o spôsob podobný YARA pravidlám, kde sa klasifikujú súbory pomocou pravidiel definovaných v LUA[1] jazyku. Na základe výsledkov týchto skriptov sa zasielajú správy na RabbitMQ server obsahujúce informácie len o type a rodine. Severita sa teda môže jednoducho odvodiť z typu vo väčšine prípadov.

Sken detekčných definícií

Informácie o detekčných definíciách je možné získavať okrem nástroja Clusty aj z ďalšieho nástroja, ktorý je primárne zameraný len na skenovanie súborov kompletnou sadou definícií. Clusty získava detekcie tiež len z tohto nástroja. Táto informácia, o vzorke a všetkých detekciách, ktoré sa nad ňou chytili, sa dá využiť viacerými spôsobmi:

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

- Ak ide o dostatočne presnú detekciu a obsahuje v názve aj informácie o triede škodlivosti, type, či rodine, môžeme ju použiť pre určenie tagu detekovaného súboru. Samotnú detekciu už ale nebudeme tagovať podľa nej samotnej.

- Veľa detekcií je generovaných automatmi a často z ich názvu sa nedá nič prečítať alebo ide len o generické informácie. Ak chceme priradiť nejaký tag tejto detekcii, tak sa môžeme pozrieť na tag detekovanej vzorky a ten použiť aj pre detekciu. Otázkou zostáva, ako sa zachovať v prípade, že jedna detekcia detekuje vzorky z naozaj rozličných rodín.

  Pri tagovaní detekcií v tomto prípade môžeme využiť už otagované vzorky, ktoré nimi boli detekované. Problémom ale môže byť to, že tag pre vzorky sa z iných nástrojov môže dostať do Taggeru oveľa neskôr. Preto je nutné implementovať odloženú aktualizáciu tagov pre tieto detekcie. Dvojice detekčná definícia – detekovaný súbor si uložíme do pomocnej tabuľky, ktorú v istých časových intervaloch budeme prechádzať a pokúsime sa aktualizovať tagy detekcií podľa príslušných súborov. Ak sa ani po istom časovom úseku nedostane do databáze tag pre súbor, bude táto dvojica z tabuľky odstránená, pretože si ich nemôžeme ukladať navždy.

AlgoHashes

Tento nástroj generuje detekčné definície a zasiela ich na RabbitMQ server spolu s informáciou o detekovanej severite, type a rodine. Táto informácia by mala byť dostatočne presná a použije sa priamo pre vytvorenie tagu danej detekčnej definície.

4.3 Porovnávanie tagov

Cieľom vyhodnocovania a prioritizovania tagov v prípade konfliktov z rôznych zdrojov je vybrať presnejší a lepší tag. Komponenta pre toto rozhodovanie (tzv. *decision maker*) je v podstate rozhodovací strom. Zdrojov informácií popísaných v predchádzajúcej sekcii je viacero a nedajú sa jednoducho zoradiť podľa priority. Niektoré poskytujú len informácie o tagu bez rodiny, iné len informáciu o severite atď. Taktiež zavedené miera dôveryhodnosti vstupuje do rozhodovania.

Po konzultácii s analytikmi boli stanovené nasledujúce pravidlá pri porovnávaní dvoch tagov:

- Manuálny tag má najvyššiu prioritu.

- YARA pravidlá majú druhú najvyššiu prioritu. Taktiež je pre ne zavedená miera dôveryhodnosti. Výnimkou v ich priorite je situácia, kedy sa porovnáva menej dôveryhodný YARA tag s AlgoHashes tagom pre súbory. Vtedy má prioritu AlgoHashes.

  Pre detekčné definície má ale AlgoHashes tag vždy prioritu pred YARA pravidlom, keďže tento nástroj priamo generuje tagy detekcií.

  V rámci porovnávania samotných YARA pravidiel sa používa komplexnejší rozhodovací algoritmus, ktorý nie je súčasťou tejto práce.

- Tag z APK Sandboxu má nižšiu prioritu.

- Nástroj Clusty má nižšiu prioritu. Implementuje niekoľko spôsobov vytvárania klasifikácii (viď 3.4.2), ktoré môžeme zoradiť podľa priorít. V rámci každej kategórie ďalej

vieme porovnávať tagy podľa ich miery dôveryhodnosti a taktiež tag s väčším počtom informácií má zvýšenú prioritu (napríklad pri porovnávaní takmer rovnakých tagov, kde len jeden z nich má rodinu).

Tento tag ďalej môže mať ale vyššiu prioritu než APK sandbox v prípade, že poskytuje informáciu o rodine a APK sandbox ju nemá.

- Tagy zo skenera detekčných definícií majú najnižšiu prioritu. Medzi sebou navzájom sa porovnávajú len podľa počtu informácií v tagu.
- V prípade rovnakých a rovnako dôveryhodných tagov sa rozhoduje už len na základe časového razítka.

4.4 API

Pre prístup k uloženým informáciám pre analytikov a iné nástroje je potrebné implementovať vhodné API. Z uložených informácií môžu byť zaujímavé hlavne nasledujúce:

- Tag pre súbor.
- Tag pre detekčnú definíciu.
- Zoznam SHA-256 hashov súborov z danej rodiny.
- Zoznam detekčných definícií s daným tagom.

V prípade zoznamu vzoriek a detekčných definícií je vhodné užívateľom vrátiť aj mieru dôveryhodnosti, s akou jednotlivé objekty majú priradený daný tag.

Pre Tagger bude vytvorené aj webové rozhranie pre generovanie a zobrazovanie štatistík, ako je popísané v ďalšej časti. Toto rozhranie je teda možné použiť aj pre vyhľadávanie a zobrazovanie tagu pre objekty.

4.5 Generovanie štatistík

V Avast Software sa ukladajú anonymizované informácie o detekovaných škodlivých súboroch u koncových užívateľov. Pre účely Taggeru sú zaujímavé len informácie o súbore, detekcii, ktorou bol detekovaný a krajine užívateľa. Z týchto informácií nie je jednoduché určiť zaujímavé analytické informácie ako šírenie konkrétneho typu či rodiny malvéru, krajiny zasiahnuté daným malvérom v posledných dňoch a pod.

Databáza ukladajúca tieto informácie, nazývaná Burger, je Hadoop[2] súborový systém (HDFS). Jedným z rozhraní pre dotazovanie sa nad jeho dátami je Hive[3]. Poskytuje rozhranie podobné jazyku SQL s vlastnými rozšíreniami a podporou pre dotazovanie sa nad distribuovanými dátami.

Štatistiky je možné jednoducho generovať prepojením informácií uložených v Taggeri a v Burger databáze. Keďže ide o obrovskú databázu v rade stoviek terabajtov, každý komplexnejší dotaz trvá minúty alebo aj desiatky minút. Z tohto dôvodu si Tagger bude ukladať tieto štatistiky do vlastnej databázy. Pre tieto účely môže byť použitá databáza pre tagy, ale nie je problém použiť úplne inú databázu. Medzi štatistikami a tagmi nie je žiadne prepojenie, ktoré by vyžadovalo ich uloženie v jednej databáze.

[2] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[3] https://hive.apache.org/

Pre účely generovania štatistík bude vytvorené webové rozhranie, v ktorom si užívateľ zadá požadované kritériá, nástroj Tagger pomocou svojich dát a dát o koncových užívateľoch vygeneruje report, uloží si ho do databázy a zobrazí užívateľovi. V prípade opakovanej požiadavky s rovnakými kritériami sa už len užívateľovi zobrazí už uložený report. Samozrejme, tagy sa môžu časom v Taggeri meniť a vygenerovaný výsledok sa teda môže taktiež mierne pozmeniť. Preto by malo byť možné na vyžiadanie užívateľom znovu vygenerovať a aktualizovať už existujúci výsledok.

Budú implementované nasledujúce typy štatistík. Kritériami, ktoré môže užívateľ zadať sú:

- Tag – severita, typ, rodina, varianta
- Minimálna hodnota dôveryhodnosti objektu (detekčných definícií) k danému tagu.
- Časový rozsah, v ktorom ho štatistiky zaujímajú.
- Úroveň ignorovania duplicitných detekcií. Ide o problém, kedy sa jeden súbor môže u jedného užívateľa detekovať niekoľkokrát denne, či už je to spôsobené úmyselne užívateľom alebo nejakým programom. Alebo o detekovanie daného súboru každý deň. Taktiež môže byť súbor detekovaný viacerými detekciami naraz.

  Tieto duplicity môžu skresľovať štatistiky a záleží na užívateľovi, o aký výsledok má záujem, čo chce tým sledovať. Či už je to celkové šírenie malvéru alebo len zistenie počtu ochránených užívateľov.

Pre uloženie štatistík do databázy postačuje jedna tabuľka, ktorá bude obsahovať všetky uvedené kritéria zadané užívateľom a výsledky štatistík. N-tica zadaných kritérií musí byť unikátna v tejto tabuľke, aby nebolo niekoľko mierne odlišných výsledkov pre rovnaké kritéria. Mal by tam byť uložený len posledný výsledok, preto je k dispozícii aj možnosť aktualizácie reportu.

Mapa zasiahnutých krajín

Na základe užívateľom zadaných kritérií sa v Taggeri vyberie zoznam všetkých vyhovujúcich detekcií. V Avast databáze sa vyberú všetky udalosti o zasiahnutých užívateľoch v zadanom časovom úseku. Následne sa spraví prienik medzi detekciami vybranými z Taggeru a detekciami z vybraných udalostí. Z výsledku sa pomocou agregačných operácií a na základe užívateľom vybranej miery definovania duplicít spočíta celkový výskyt detekovania škodlivých súborov v jednotlivých krajinách. Analytikovi sa pomocou webového rozhrania zobrazí mapa, na ktorej môže vidieť počet incidentov v jednotlivých krajinách.

Graf denných detekcií

Na základe užívateľom zadaných kritérií sa v Taggeri vyberie zoznam všetkých vyhovujúcich detekcií. V Avast databáze sa vyberú všetky udalosti o zasiahnutých užívateľoch v zadanom časovom úseku. Následne sa spraví prienik medzi detekciami vybranými z Taggeru a detekciami z vybraných udalostí. Z výsledku sa pomocou agregačných operácií a na základe užívateľom vybranej miery definovania duplicít spočíta výskyt detekovania škodlivých súborov v jednotlivých dňoch daného časového úseku. Analytikovi sa pomocou webového rozhrania zobrazí graf, na ktorom môže vidieť počet incidentov v jednotlivých dňoch.

Zoznam najčastejších rodín

Tento typ reportu sa odlišuje od predchádzajúcich. Negeneruje sa, samozrejme, v prípade, že užívateľ v rámci kritérií zadal rodinu, ktorá ho zaujíma. Vytvorí sa len v prípade chýbajúcej rodiny a môže poskytovať informáciu o populárnosti rodín daného typu.

V tomto prípade je teda aj postup vytvorenia reportu mierne odlišný. Z Tagger databázy sa vytiahne zoznam všetkých detekcií vyhovujúcich zadaným kritériám. V Avast databáze sa vyberú všetky udalosti o zasiahnutých užívateľoch v danom časovom úseku. Spraví sa prienik medzi detekciami z Taggeru a detekciami z vybraných udalostí. Vo výsledku sa následne spočíta celkový počet výskytov jednotlivých detekcií. Tento zoznam počtu výskytov detekcií sa vráti späť do Taggeru, kde sa dohľadá tag týchto detekcií. Počet výskytov týchto detekcií sa použije pre sčítanie výskytov jednotlivých rodín z tagov priradených týmto detekciám.

Sumárne štatistiky

Okrem štatistík uvedených v predchádzajúcej časti je zaujímave z hľadiska analytikov vidieť aj sumárne štatistiky jednotlivých severít. Ide len o report, ktorý zobrazuje celkový výskyt napríklad malvéru a k tomu aj podrobnejšie rozdelenie na jednotlivé typy. Teoreticky sa toto neodlišuje nijak výrazne od predchádzajúcej časti. Užívateľ môže vo webovom rozhraní zvoliť kritérium *malvér* a odoslať dotaz. Naviac ale vidí zastúpenie jednotlivých typov zvolenej severity a môže si pre každý typ zobraziť jej štatistiky, ako uvádza predchádzajúca časť. Ide skôr len o spôsob prezentácie štatistík vo webovom rozhraní, než rozširovanie analytických dotazov.

Tento typ štatistík je teda možné vygenerovať ako súčet štatistík pre jednotlivé typy danej severity. V databáze môže byť uložené mapovanie sumárneho výsledku k jeho jednotlivým podvýsledkom a na aplikačnej úrovni sa tieto podvýsledky už len sčítajú. Ukladanie výsledkov sumárnych štatistík by znamenalo ich aktualizáciu v prípade aktualizácie akéhokoľvek podvýsledku. Preto môže byť jednoduchšie to spočítať na aplikačnej úrovni, keďže ide len o najviac pár desiatok podvýsledkov.

Kapitola 5

Rozšírenie nástroja Clusty

V rámci tejto práce je potrebné rozšíriť aj nástroj Clusty a umožniť tak získavanie dát vyžadovaných nástrojom Tagger a zjednodušiť ich interakciu. Návrh, implementácia a testovanie vylepšení sú popísané v nasledujúcich sekciách.

5.1 Návrh

Nie všetky informácie z nástroja Clusty popísané v sekcii 4.2 sú priamo k dispozícii pre Tagger. Sú uložené v jeho internej databáze, ale niektoré nie sú dostupné pre vonkajšie použitie.

Tagger navyše potrebuje pracovať s nejakou hodnotou určujúcou mieru dôveryhodnosti daného tagu pre ich lepšie prioritizovanie a vyhodnocovanie v prípade konfliktov z rôznych zdrojov, či aj konfliktov dvoch tagov pre jeden objekt z nástroja Clusty.

V automatickej klasifikácii implementovanej v práci [27] sa s postupom času ukázali rôzne malé nedostatky, či skôr priestory na zlepšenie.

Publikovanie správy do RabbitMQ pre manuálne klasifikácie zhlukov

Manuálne klasifikácie môžu byť vytvorené len analytikmi (cez webové rozhranie). Aktuálna implementácia zasiela tieto správy len pre automatické klasifikácie (viď sekcia 3.4.2). Teda žiaden externý nástroj sa nikdy nedozvie o manuálne klasifikovanom zhluku, čo znamená stratu informácií pre Tagger. Manuálne klasifikácie sú považované za dostatočne dôveryhodný zdroj tagov (má najvyššiu hodnotu dôveryhodnosti v nástroji Clusty), preto by táto informácia nemala byť zahodená.

Publikovanie správ o nových (len automatických) klasifikáciách je už v nástroji Clusty implementované. V rámci tejto práce je to len rozšírené o zasielanie ďalšej správy v prípade manuálnej klasifikácie.

Prístup k zoznamu Avast detekcií v zhluku

Každý zhluk v nástroji Clusty má svoj tag (klasifikáciu) a všetky jeho vzorky obsahujú zoznam Avast detekcií, ktoré ich detekovali. Pre nástroj Tagger je užitočnou informáciou daný tag a zoznam všetkých týchto detekcií. V rámci tohto rozšírenia je potrebné pridať podporu do verejného API. Nová URL adresa (tzv. *endpoint*) pre zadané ID zhluku vráti zoznam unikátnych detekcií a ich percentuálne zastúpenie v rámci zhluku. Ide o hodnotu udávajúcu, koľko percent vzoriek tohto zhluku je detekovaných danou detekciou.

Redukcia množstva publikovaných správ

Do RabbitMQ sú publikované správy pri aktualizácii klasifikácii zhluku, pridaní vzoriek do zhluku a pod. Pri analýze nástroja Clusty v rámci tejto práce bol odhalený problém s nadmerným množstvom zasielaných správ o klasifikovaní zhlukov, ktoré len opakujú informácie z predchádzajúcich. Kvôli aj tak už dosť veľkému množstvu očakávaných správ, ktoré bude musieť Tagger spracovávať, bude vhodné opraviť tento problém priamo v nástroji Clusty, keďže duplikované správy len s novším časovým razítkom nenesú žiadnu novú informáciu.

Dôvodom je to, že klasifikácia obsahuje viac informácií, než len položky potrebné pre tag (teda rodina, severita, ...). Clusty si ukladá aj metadata s informáciami, na základe ktorých bola klasifikácia vytvorená. Tieto informácie ale nie sú potrebné v publikovaných správach. Preto je nutné upraviť rozhodovanie o zasielaní správ tak, aby každá nová správa pre daný zhluk sa líšila od predchádzajúcej.

Rozšírenie automatických klasifikácií nástroja Clusty o hodnotu dôveryhodnosti

Automatické klasifikácie v nástroji Clusty používajú viaceré zdroje informácií a heuristiky, ako je popísané v časti 3.4.2. Nie všetky zdroje a metódy majú ale rovnakú presnosť. Analytici, ale aj ďalšie nástroje pracujúce s klasifikovanými zhlukmi (napr. automatický generátor detekčných definícií), môžu využiť informáciu o spoľahlivosti/dôveryhodnosti klasifikácie. Je potrebné teda zaviesť vhodnú mieru udávajúcu, ako veľmi môžeme veriť danej automatickej klasifikácii.

Jednoduchou možnosťou je použitie percentuálneho vyjadrenia dôveryhodnosti. Pre metódy klasifikovania popísané v časti 3.4.2 môžeme zaviesť výpočet dôveryhodnosti nasledujúcim spôsobom:

- klasifikácia podľa YARA pravidiel – YARA pravidlá obsahujú v metainformáciách položku označujúcu spoľahlivosť daného pravidla. Ide len o ordinálny atribút. Spoľahlivosť takejto klasifikácie teda môžeme vyjadriť priradením pevnej percentuálnej hodnoty každej ordinálnej hodnote (percentá sú zvolené po dohode s autormi pravidiel).

- klasifikácie podľa antivírusových detekcií – v tomto prípade môžeme už zaviesť výpočet dôveryhodnosti na základe množstva a kvality dostupných informácií. Keďže môžu vzniknúť dva rôzne typy klasifikácii (čisté vzorky, škodlivé vzorky – zahŕňajú zvyšok, teda severity malware, PUP, tool), aj výpočet sa bude líšiť pre oba typy:

  - zhluk čistých vzoriek – zhluk je označený ako *clean* na základe celkového počtu detekcií nad vzorkami. U každej vzorky môžu nastať pre každý tri prípady: vzorka detekovaná ako škodlivá, vzorka označená antivírusom ako čistá, vzorka nebola daným antivírusom otestovaná. Ďalej budeme hovoriť teda o škodlivej, čistej a neznámej detekcii.

    Pre výpočet dôveryhodnosti použijeme zastúpenie týchto troch kategórií. Výsledná hodnota dôveryhodnosti je z nich zložená takto:

    1. Celková hodnota nesmie presiahnuť hodnotu $T = 100 - U$, kde U je percentuálne zastúpenie vzoriek, ktoré neboli testované žiadnym antivírusom.
    2. Hodnota T je ďalej znížená o pomer všetkých škodlivých detekcií nad vzorkami ku všetkým detekciám (teda škodlivým aj čistým).

3. Škodlivé detekcie od rôznych antivírusov ale majú rôznu váhu. V nástroji Clusty sú rôznym antivírusom priradené rôzne kategórie dôveryhodnosti (nesúvisiace s dôveryhodnosťou ku klasifikáciám popisovanou v tejto časti). Každá kategória má svoju váhu.
4. To, ako škodlivá detekcia od daného antivírusu zníži hodnotu T teda závisí na kategórii antivírusu. Výsledný výpočet je teda takýto:

$$A = T\left(1 - \frac{100}{D}\sum_{k \in K} w_k S_k\right)$$

kde K je množina všetkých kategórií antivírusov, w_k je váha kategórie k, S_k je počet škodlivých detekcií všetkých antivírusov z kategórie k a D je celkový počet všetkých škodlivých aj čistých detekcií (bez ohľadu na kategóriu).
5. Výsledná hodnota je v percentách. Následne je ale normalizovaná do intervalu $\langle 0, 95\rangle$, pretože ide predsa len o automatický klasifikátor a vyššie hodnoty sú prenechané dôveryhodnejším zdrojom klasifikácií.

– zhluk škodlivých vzoriek – zhluk je označený ako škodlivý na základe celkového počtu detekcií nad vzorkami. Rovnako ako bolo uvedené pre čisté zhluky, pracujeme s troma typmi detekcií – škodlivé, čisté a neznáme.

Výpočet dôveryhodnosti je nasledovný:

1. Celková hodnota nesmie presiahnuť hodnotu $T = 100 - U - S$, kde U je percentuálne zastúpenie vzoriek, ktoré neboli testované žiadnym antivírusom. S je percentuálne zastúpenie čistých vzoriek.
2. T je ďalej znížené o počet čistých detekcií nad vzorkami. Teraz už nejde o čisté vzorky, kedy musia byť všetky jej detekcie čisté. Hovoríme už len o vzorkách, ktoré boli niektorými antivírusmi detekované ako škodlivé a inými ako čisté. Ide len o zníženie v pomere čistých detekcií ku všetkým:

$$N = T - 100\frac{C}{D}$$

kde C je počet čistých detekcií a D je počet všetkých čistých aj škodlivých detekcií.
3. Táto hodnota sa použije pre celkový výpočet dôveryhodnosti:

$$A = 100\frac{N}{D}\sum_{k \in K} w_k S_k$$

kde K je množina všetkých kategórií antivírusov, w_k je váha kategórie k, S_k je počet škodlivých detekcií všetkých antivírusov z kategórie k a D je celkový počet všetkých škodlivých aj čistých detekcií (bez ohľadu na kategóriu).
4. Výsledná hodnota je v percentách. Následne je ale normalizovaná do intervalu $\langle 0, 95\rangle$, pretože ide predsa len o automatický klasifikátor a vyššie hodnoty sú prenechané dôveryhodnejším zdrojom klasifikácií.

Vylepšenie detekcie názvu rodiny v antivírusových detekciách

Ako je už popísané v časti 3.4.2, jedným z problémov pri získavaní názvu rodiny z antivírusových detekcií je používanie generických názvov rodín. Popísaný algoritmus už filtruje názvy rodín pomocou známeho zoznamu generických, používa heuristiky pre rozlíšenie rodiny od hashu vzorky, detekcie, či inej informácie, uvedenej v názve detekcie. Ďalej je u niektorých antivírusov možné jednoznačne určiť (syntakticky), ktorá časť detekcie označuje rodinu. V generickom parseri detekcií je každý token, ktorý neoznačuje typ ani severitu, označený ako potenciálny názov rodiny, podrobnejšie sú špecifické a generické parsery popísané v [27].

V nástroji Clusty bolo v rámci tejto práce objavených viacero klasifikácií so zlými rodinami, pričom po analýze zhlukov sa zistilo, že je to spôsobné hlavne generickým parserom. Problémom je to, že všetko, čo nie je typom ani severitou je potenciálne rodina. V tejto práci je teda implementované vylepšenie, ktoré sa snaží eliminovať množstvo extrahovaných neplatných rodín. Ide o nasledujúcu heuristiku.

- Väčšina týchto analyzovaných neplatných rodín boli krátke reťazce, často boli celé veľkými písmenami, začínali číslicou, prípadne obsahovali viacero čísel.
- Takéto tokeny sa v generickom parseri označia za pravdepodobných kandidátov na rodinu.
- Po analýze všetkých detekcií sa z pravdepodobných kandidátov pridajú k tým skutočným kandidátom len tie, ktoré sa už nachádzajú medzi skutočnými. Zvyšné sa zahodia. Skutočnými kandidátmi sú tokeny zo špecifických parserov a tokeny, u ktorých si aj generický parser bol dostatočne istý, že ide o rodinu.

  Motivácia je taká, že ak daný token už bol už niektorým zo špecifických (presnejších) parserov označený ako kandidát na rodinu, tak ho zarátame aj zo zoznamu pravdepodobných rodín, aby sme nestratili informáciu o celkovom počte výskytov daného tokenu, ktorá je následne potrebná pre klasifikátor.

5.2 Implementácia

Všetky tieto rozšírenia sú implementované v Python časti nástroja Clusty. Ide o verziu 3.7. V rámci tejto časti práce ide len o úpravu existujúceho kódu a rozšírenie funkcionalít. Spôsob implementácie týchto rozšírení je teda ovplyvnený už existujúcim kódom a je v súlade s architektúrou nástroja Clusty.

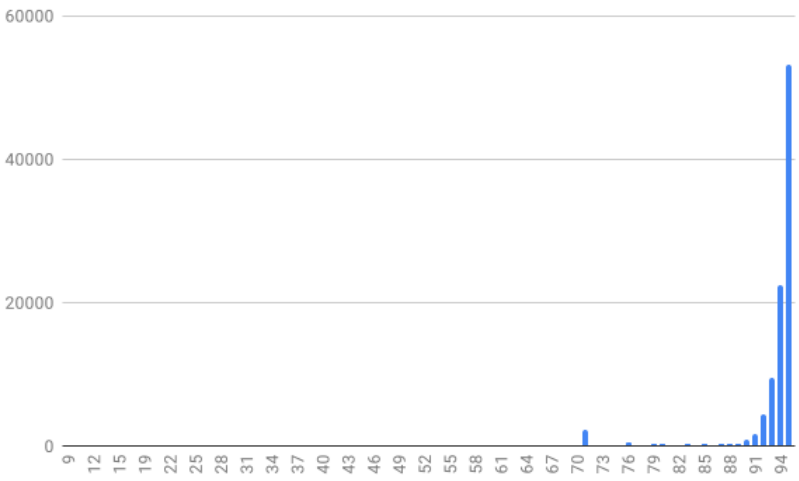
5.3 Testovanie

Všetky rozšírenia nástroja Clusty boli dôsledne otestované. Celkovou sadou 40 jednotkových a integračných testov je pokrytý kód všetkých implementovaných vylepšení. Okrem toho bolo nutné upraviť aj už existujúce testy, ktoré nepočítali s novou funkcionalitou.

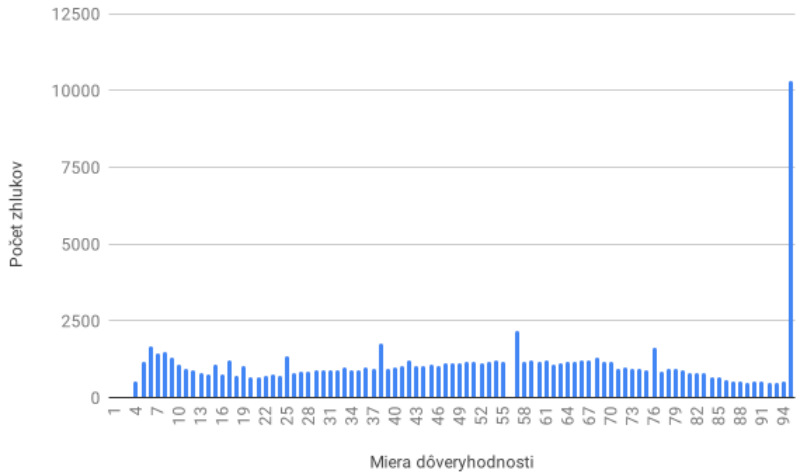
Nástroj Clusty je v tejto práci len rozširovaný a po celý čas bežal v produkčnej verzii. Tieto rozšírenia sa teda priebežne dostávali do produkčnej verzie a ich implementácie je už teda otestovaná aj v reálnom nasadení.

Okrem týchto testov boli pre výpočet miery dôveryhodnosti klasifikácii vygenerované histogramy dôveryhodnosti v produkčnom nasadení. Histogram hodnôt miery dôveryhodnosti vypočítaný pre zhluky čistých vzoriek je na obrázku 5.1 a histogram pre zhluky

škodlivých súborov je na obrázku 5.2. Tieto grafy sčasti sú závislé na kvalite zhlukov vytvorených nástrojom Clusty. Manuálnou analýzou náhodnej časti týchto výsledkov sa tieto výsledky ukazujú ako vhodné pre určovanie dôveryhodnosti.



Obr. 5.1: Histogram miery dôveryhodnosti 100 000 zhlukov čistých vzoriek



Obr. 5.2: Histogram miery dôveryhodnosti 100 000 zhlukov škodlivých vzoriek

Kapitola 6

Implementácia platformy

Ako je popísané v kapitole 4, nástroj Tagger môžeme rozdeliť do niekoľkých oddelených celkov. Ide o databázu, aplikáciu pre odoberanie nových informácií z externých nástrojov, modul pre rozhodovanie a výber najlepšieho tagu zo všetkých dostupných informácií a webové a aplikačné rozhranie pre prístup k uloženým informáciám a generovanie štatistík.

Pre ukladanie informácií bola zvolená PostgreSQL databáza (viď sekciu 4.1) a pre zvyšné časti nástroja bol zvolený jazyk Python, verzia 3.7 (posledná stabilná verzia v čase práce na projekte). Hlavným dôvodom pre Python je možnosť využiť distribuovaný framework Celery. Kritickou časťou nástroja nie je spracovanie jednej informácie (prijatie tagu, vyhodnotenie, porovnanie s inými zdrojmi atď.), nejde o výpočtovo náročné operácie. Problémom je zvládnutie veľkého množstva prichádzajúcich informácií. Pričom každá nová informácia zahŕňa komunikáciu s databázou, vyhľadávanie v nej, získavanie dodatočných informácií z externých nástrojov – sieťová komunikácia, vyhľadanie. Distribuovaný systém ako Celery, s veľmi jednoduchou podporou škálovateľnosti umožní jednoduché prispôsobovanie sa zmenám v množstve správ.

Ďalšou výhodou jazyka Python je veľké množstvo podporných knižníc. Dôležité sú hlavne knižnice podporujúce AMQP protokol pre komunikáciu s RabbitMQ serverom, knižnice pre spojenie s Burger databázou, či SSH spojenie a prenos dát.

Jednotlivé časti platformy sú implementované nasledovne.

6.1 Databáza

Pre komunikáciu s databázou je použitá knižnica v jazyku Python – SQLAlchemy[1]. Poskytuje klasické SQL rozhranie (tzv. *SQLAlchemy Core*) aj objektovo-relačné mapovanie (ORM). Pre lepší výkon na aplikačnej úrovni je ale použité len SQLAlchemy Core rozhranie. ORM poskytuje vysokoúrovňové rozhranie, ale môže byť pomalšie kvôli mapovaniu na objekty. Nie je určené pre vysokovýkonné hromadné vkladanie dát [21].

Ako adaptér (angl. *driver*) medzi SQLAlchemy a databázou je použitý Psycopg2[2]. Je implementovaný hlavne v jazyku C a ide o obálku nad `libpq`[3] knižnicou – rozhranie pre komunikáciu s PostgreSQL databázou.

[1] https://www.sqlalchemy.org/
[2] https://pypi.org/project/psycopg2/
[3] https://www.postgresql.org/docs/current/libpq.html

Psycopg2 je zameraný na vysoký výkon, bezpečnosť a viac-vláknové aplikácie s veľkým množstvom paralelných zápisov, aktualizácií dát a častým vytváraním a rušením databázových kurzorov.

6.1.1 Ukladanie dát

Pre prácu s dátami v nástroji Tagger je implementovaná v module `tagger.tagger_db` trieda `TaggerDB`, ktorá poskytuje plné implementačné rozhranie pre prácu s tagovanými objektmi a tagmi – vkladanie, aktualizáciu, čítanie. Kvôli paralelnému spracovávaniu prichádzajúcich udalostí a vyhodnocovaniu priorít tagov na aplikačnej úrovni je nutné využiť zamykanie aktuálne spracovávaných objektov. Pri porovnávaní priorít dvoch tagov (jeden je nový, z externých nástrojov, druhý je aktuálne uložený v databáze) je nutné načítať aktuálny tag, porovnať s novým a prípadne zapísať nový. Medzitým ale mohol iný proces zapísať ďalší tag a teda toto vyhodnotenie na aplikačnej úrovni by už bolo bezvýznamné a prepis dát by mohol byť chybou. Preto sa využíva SQL klauzula `SELECT ... FOR UPDATE`. Takto zamknutý riadok je k dispozícii na čítanie (`SELECT`, bez doložky `FOR UPDATE`), ale nemôže ho iný proces modifikovať, zmazať, ani čítať so zámkom (`SELECT ... FOR UPDATE`) po dobu zamknutia.

6.1.2 Ukladanie vygenerovaných štatistík

Štatistiky pre analytické účely nemusia byť súčasťou databázy pre ukladanie tagov. NoSQL databázy môžu poskytnúť jednoduchšiu prácu s nimi, ale prináša to nutnosť spravovania ďalšej databázy.

Preto je aj pre tieto dáta použitá tá istá PostgreSQL databáza. Vzhľadom na nie tak veľké množstvo dát a ich jednoduchú schému nie je problém toto rozhodnutie neskôr zmeniť.

Pre ukladanie a prístup k týmto dátam je implementovaná trieda `WithDetHitsResult` (ide o triedu len s metódami, tzv. *mixin*) v module `tagger.det_hits`, ktorý rozširuje funkcionalitu triedy `TaggerDB`, keďže reporty sa ukladajú v rovnakej databáze ako tagy.

Okrem informácií ukladaných k štatistikám uvedeným v sekcii 4.5 môže byť užitočné ukladať si čas ich vytvorenia a pomocný status daného reportu. Čas je samozrejme len pre informovanosť užívateľa. Status je pomocná informácia pre analytikov a taktiež pre nástroj Tagger. Analytici môžu vidieť, či ich požiadavka čaká na spracovanie, je aktuálne spracovávaná alebo je už dokončená. Z implementačného hľadiska je status dobrý pre zabránenie zbytočnému a časovo náročnému generovaniu štatistík pre rovnaké kritériá viacerými procesmi súčasne. Ako bude uvedené neskôr, požiadavky z webového rozhrania sa spracovávajú paralelne viacerými procesmi.

6.2 Prijímanie správ z externých nástrojov

Všetky externé nástroje, z ktorých Tagger čerpá informácie, zasielajú správy na RabbitMQ server. Získavanie týchto správ je implementované pomocou klientskej knižnice Pika[4]. Pre každý externý nástroj je spustený dedikovaný proces, ktorý len konzumuje prichádzajúce správy. Niektoré nástroje produkujú viacero typov správ odlíšených pomocou smerovacieho kľúča. Pre každú z nich sa v rámci procesu vytvorí vlastné vlákno.

Prijaté správy sú následne spracovávané takto:

[4] https://pypi.org/project/pika/

- Analyzuje sa obsah správy. V niektorých prípadoch neobsahuje informácie zaujímavé pre Tagger. Takéto správy sú zahodené.
- Zo správy s užitočnými informáciami pre Tagger sa vyberú dôležité informácie (často toho obsahujú viac, než je potrebné) a odošle sa nová správa do interného RabbitMQ serveru dedikovaného pre Tagger.
- V prípade problému pri odoslaní správy do interného RabbitMQ sa odosielanie opakuje až pokým nebude úspešné. Správ sa teda môžu začať hromadiť vo fronte.
- Po úspešnom odoslaní správy sa zašle potvrdenie na RabbitMQ odkiaľ bola prijatá a spracuje sa ďalšia správa.

Ide teda len o preklad správ (a prípadné kontroly) do internej podoby pre Tagger. Nedochádza tam k žiadnej interakcii s databázou ani s tagmi. Tieto správy sú následne zaslané na interný RabbitMQ server, na ktorý je napojený Celery framework. Použitie tohto frameworku umožňuje veľmi jednoducho riešiť problém škálovateľnosti aplikačnej úrovne.

Správy z interného RabbitMQ už spracováva veľké množstvo procesov vytvorených pomocou Celery. Väčšina správ obsahuje kompletné informácie potrebné pre vytvorenie či aktualizáciu tagu. V niektorých prípadoch je ale nutné zisťovať z externých nástrojov ešte dodatočné informácie pomocou ich API. Potom sa už môže spustiť aktualizácia tagu pre objekt.

Porovnávanie tagov a ich prioritizovanie je implementované ako rozhodovací strom, kde pravidlá hľadia hlavne na zdroj tagu a množstvo obsiahnutých informácií (nie každý nástroj poskytuje kompletné informácie pre tag, ako boli vymenované v sekcii 4.1).

6.3 Generovanie štatistík

Ako je popísané v sekcii 4.5, dotazy pre generovanie štatistík vyžadujú na vstupe detekcie z Tagger databázy. V prípade detekcií majúcich tag konkrétnej rodiny môžeme hovoriť o tisícoch, prípadne desaťtisícoch detekcií. Počet detekcií obecne pre daný typ malvéru môže siahať až do jednotiek milónov. V tomto prípade je nutné vyriešiť spôsob prieniku Tagger detekcií a Burger detekcií. Pri tisícoch detekcií je možné napísať dotaz pomocu klauzuly `IN`, ktorým vyberieme z Burger detekcií len tie potrebné. Pri státisícoch ale už dostávame obrovský dotaz a veľmi náročný na spracovanie. HDFS ale umožňuje vytváranie ďalších vlastných tabuliek pre užívateľov, ktoré je možné následne v dotazoch prepojiť (`JOIN`) s ostatnými tabuľkami. Taktiež poskytuje rozhranie pre programové spúšťanie dotazov – JDBC[5] aj ODBC[6].

Pomocou nich môžeme teda aj z jazyka Python pristupovať k HDFS dátam. Pre Tagger bolo zvolené ODBC rozhranie, ktoré je implementované v knižnici `pyodbc`[7]. Táto knižnica ale vyžaduje HIVE–ODBC[8] adaptér.

Dotazy pre generovanie štatistík sú teda spúšťané pomocou ODBC rozhrania. Pre nahranie detekcií do tabuliek na HDFS ale môžeme využiť jednoduchší prístup, a to pomocou SSH spojenia. Pomocou `scp`[9] a `paramiko`[10] knižníc môžeme z Python kódu nahrať dáta na

[5] https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html
[6] https://docs.microsoft.com/en-us/sql/odbc/reference/what-is-odbc?view=sql-server-2017
[7] https://github.com/mkleehammer/pyodbc
[8] https://www.progress.com/odbc/hortonworks-hive
[9] https://pypi.org/project/scp/
[10] http://www.paramiko.org/

server, umiestniť do HDFS adresára a následne už len pomocou ODBC vytvoríme tabuľku s týmito dátami. Dôvodom použitia SSH spojenia namiesto ODBC je hlavne to, že týchto detekcií je veľké množstvo, státisíce aj milióny. Dotazy cez ODBC pre nahranie tak veľkého množstva detekcií sú časovo nezvládnuteľné.

V prípade dotazov pre šírenie škodlivých súborov v jednotlivých krajinách alebo denných štatistikách je výsledok dotazu celkom malý – limitovaný počtom dní požadovaného rozsahu alebo počtom krajín sveta. Takýto malý výsledok opäť môžeme jednoducho získať späť cez ODBC. Pri zisťovaní najčastejšie sa vyskytujúcich rodín je ale výsledkom počet výskytov všetkých detekcií nahraných z Tagger databázy. V tomto prípade bude opäť lepšie stiahnuť výsledok cez SSH spojenie. Výhodou je, že v prípade zlyhania prenosu tak veľa dát môžeme SSH prenos zopakovať. Pri ODBC spojení by bolo nutné znova spustiť celý dotaz. Na strane Hive je taktiež jednoduchšie uložiť výsledok do tabuľky.

6.4 Distribuované spracovanie úloh

Ako popisuje sekcia 6.2, správy z externých nástrojov sú preložené do internej podoby a zaslané na RabbitMQ server dedikovaný pre Tagger. Na ten je napojený framework Celery, ktorý spravuje vybraný počet paralelných procesov konzumujúcich a vykonávajúcich tieto interné správy. Veľkou výhodou Celery je možnosť spustiť ho dynamicky na viacerých serveroch a dynamicky upravovať počet jeho procesov. Ide o veľmi jednoduchú možnosť škálovania.

Správy z rôznych služieb sú rozdelené do vlastných front na internom RabbitMQ serveri. Každá fronta je pre jednu konkrétnu úlohu. Ide o nasledujúce úlohy implementované v module `tagger.tasks`:

- Pridanie tagu detekčných definícií z nástroja AlgoHashes.
- Pridanie či aktualizácia tagu detekčnej definície podľa súbory, ktorý ňou bol detekovaný pri skenovaní v externom nástroji.
- Pridanie či aktualizácia informácií o vzorkách a detekciách zo zhluku nástroja Clusty.
- Pridanie či aktualizácia tagu súboru pridaného do zhluku v nástroji Clusty.
- Pridanie či aktualizácia tagu detekčných definícií pri ich aktualizácií v nástroji Clusty.
- Pridanie či aktualizácia tagu súboru podľa YARA pravidla.
- Pridanie či aktualizácia tagu súboru podľa výsledku zo sandboxu pre APK súbory.
- Pridanie či aktualizácia tagu súboru podľa detekcie, ktorá ho detekovala.

Všetky úlohy by mohli byť zasielané do jedinej fronty. Výhodou oddelených front je ale lepší prehľad o počte správ daného typu, prípadne možnosť dedikovať konkrétne množstvo procesov ku konkrétnej fronte (neimplementované v rámci tejto práce).

Ďalšie využitie Celery procesov je pre generovanie štatistík. Ide o oddelenú skupinu procesov, ktorá vybavuje požiadavky na generovanie štatistík zadané užívateľmi vo webovom rozhraní, ako popisuje sekcia 6.3.

Tretie využitie frameworku Celery je periodická aktualizácia tagov pre detekčné definície. Ako popisuje sekcia 4.2, detekcii môžeme priradiť tag podľa nej detekovaného súboru. Lenže tag tohto súboru sa môže do Taggeru dostať neskôr, než informácie o detekcii súboru danou detekčnou definíciou. Preto sa využíva osobitná tabuľka s mapovaním detekcií na detekované súbory. Obsahuje ešte aj časové razítko vloženia záznamu pre jeho prípadné odstránenie po vypršaní limitu pre uloženie v tejto tabuľke.

S využitím Celery Beat[11] komponenty pre periodické úlohy sa informácie z tejto tabuľky v pravidelných intervaloch zasielajú na aktualizáciu. Ak sa tag pre vzorku už nachádza v databáze, tak sa použije pre danú detekciu. Po určitom časovom úseku sa mapovanie z tabuľky zahodí, pretože nie je možné ukladať si navždy všetky informácie a pokúšať sa všetky aktualizovať.

6.5 Webové rozhranie a API

Pre implementáciu webového rozhrania je zvolený framework Flask[12] s webovým serverom Gunicorn[13]. Flask je populárny framework, ktorý spolu s Jinja2[14] knižnicou s webovými šablónami je jednoduchý, flexibilný a podporovanými mnohými knižnicami (napr. SQLAlchemy). Gunicorn je WSGI (Web Server Gateway Interface – rozhranie medzi webovým serverom a webovou aplikáciou) server s jednoduchou konfiguráciou pre Python, automatickou správou procesov a jednoduchou rozšíriteľnosťou. Pre vytvorenie webového rozhrania je ďalej použité CSS, JavaScript (JS), knižnica JQuery[15] a knižnica Bootstrap[16] obsahujúca šablóny pre jednoduchší vývoj v HTML, CSS a JS.

Pomocou týchto nástrojov je implementované webové rozhranie aj API. Obe časti sú popísané v nasledujúcich sekciách.

6.5.1 Webové rozhranie

Základom webového rozhrania je formulár pre generovanie štatistík o útokoch škodlivých súborov u koncových užívateľov, viď 6.1. Užívateľ si vyberie kritéria týkajúce sa tagu (musí si zvoliť aspoň jedno) a kritéria pre časový rozsah útokov. Po odoslaní dotazu sa na serveri spustí jeho spracovanie v Celery procese a užívateľ bude presmerovaný na stránku s výsledkom (spočiatku prázdnu, viď 6.2), ktorá bude aktualizovaná po vygenerovaní štatistík.

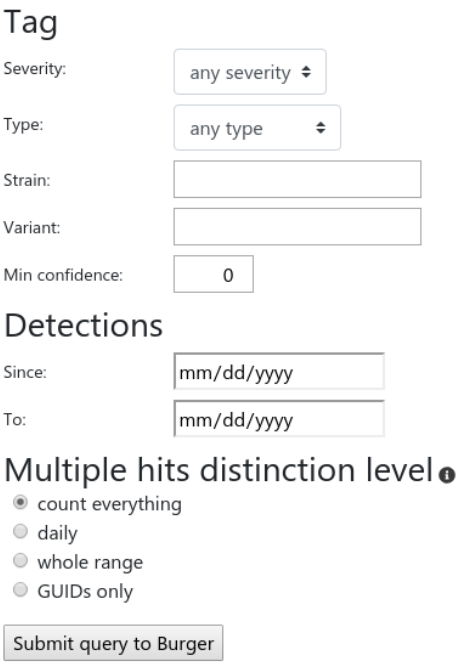
Všetky štatistiky sú generované sekvenčne pre zníženie záťaže na Burger servery. Ide teda o vygenerovanie mapy útokov v jednotlivých krajinách, grafu denných útokov, prípadne najčastejšie rodiny. Každá štatistika je zobrazená akonáhle sa získa jej výsledok. Ukážka mapy útokov je na obrázku 6.4 a ukážka grafu denných útokov je na obrázku 6.3.

Okrem generovania štatistík môže užívateľ využiť webové rozhranie aj pre vyhľadávania tagov pre objekty. Zadaním identifikátora objektu do vyhľadávacieho políčka, ktoré sa snaží rozpoznať typ vstupu, či ide o hash súboru alebo detekčnú definíciu, sa užívateľovi zobrazí tag daného objektu. Ide len o lepšiu webovú reprezentáciu tagu rovnakého ako z API (viď API výstup v ukážke 6.5).

6.5.2 API

Pre prístup k informáciám uloženým v databáza nástroja Tagger je implementované aplikačné rozhranie s využitím spomínanej architektúry webového rozhrania – teda pomocou knižnice `flask`. Hlavnými výhodami je jednoduchosť rozhrania, univerzálne rozhranie pre všetkých užívateľov, oddelenie užívateľov od uloženia dát v databáze s možnosťou zmeny

[11] https://docs.celeryproject.org/en/latest/userguide/periodic-tasks.html
[12] http://flask.pocoo.org
[13] https://gunicorn.org
[14] http://jinja.pocoo.org
[15] https://jquery.com
[16] https://getbootstrap.com



Obr. 6.1: Webový formulár pre vygenerovanie štatistík útokov škodlivých súborov.

jej schémy bez vplyvu na užívateľov, široká podpora JSON formátu v programovacích jazykoch.

URL adresy API rozhrania popísané v sekcii 4.4 sú implementované nasledovne:

- Tag pre súbor – Vstupom je SHA-256 hash súboru.

  URL: `/api/sample_sha256/<SHA-256-hash>/tag/`.

- Tag pre detekciu – Vstupom je názov detekcie.

  URL: `/api/avast_detection/<názov-detekcie>/tag/`.

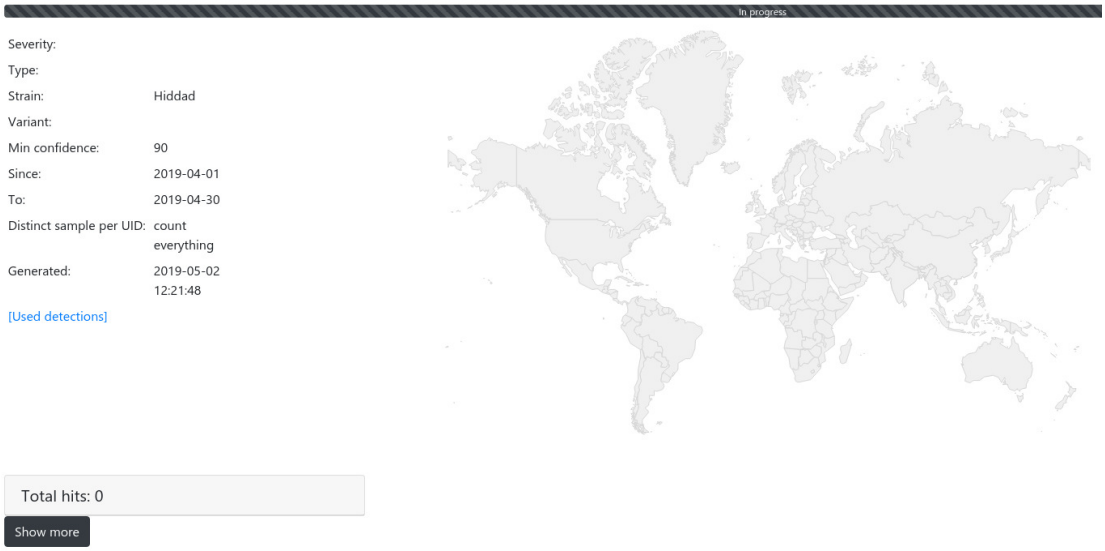
- Zoznam SHA-256 hashov súborov z danej rodiny – Vstupom je názov rodiny, výstupom je zoznam súborov a miera dôveryhodnosti priradenému tagu (rodine) ku konkrétnemu súboru.

  URL: `/api/strain/<rodina>/samples_sha256/`.

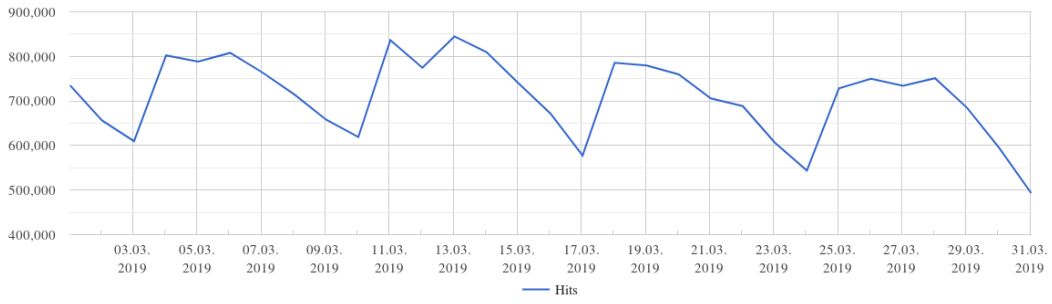
- Zoznam detekcií s daným tagom – Vstupom je tag (niektoré jeho položky) a výstupom je zoznam detekcií a miera dôveryhodnosti im priradenému tagu.

  URL: `/api/tag/avast_detection_definitions/`.

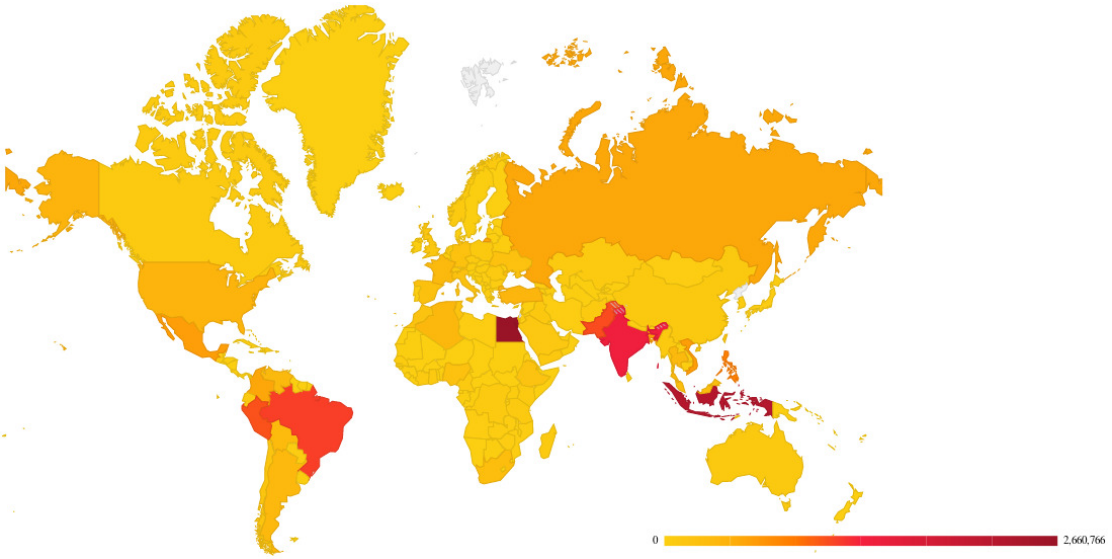
Výstupom pre tag súboru a detekcie je tag vo formáte ako je zobrazený v ukážke 6.5. V prípade, že sa daný objekt v Taggeri nenachádza, výstupom je opäť JSON s chybovou správou a chybovým kódom.



Obr. 6.2: Stránka po odoslaní dotazu na vygenerovanie štatistík čakajúca na výsledky.



Obr. 6.3: Ukážka denného výskytu incidentov škodlivých súborov u užívateľov pre istý typ malvéru



Obr. 6.4: Ukážka mapy počtu incidentov škodlivých súborov u užívateľov pre istý typ malvéru


```
{
  "confidence": 99,
  "platform": "pe",
  "severity": "malware",
  "source": {
    "detail": "by YARA rule",
    "service": "clusty"
  },
  "strain": "DownloadGuide",
  "timestamp": "2018-10-12T06:11:29.000000Z",
  "type": "adware",
  "variant": null
}
```


Kód 6.5: Ukážka výsledku API dotazu na tag súboru.

Kapitola 7

Testovanie

Pri vývoji nástroja Tagger je nutné testovať hneď niekoľko rôznych aspektov. Základom testovania implementácie sú jednotkové a integračné testy. Ide teda hlavne o syntaktickú a sémantickú kontrolu implementácie, prevenciu pred nechceným pozmenením existujúcej funkcionality atď.

Ďalej je nutné otestovať výkonnosť celej platformy. Teda rýchlosť reakcie na prichádzajúce udalosti z externých služieb, rýchlosť aktualizácie tagov v databáze a rýchlosť odpovedí na užívateľské požiadavky z webové rozhrania a API.

Problematickejšou časťou je testovanie kvality rozhodovacieho stromu, ktorý prioritizuje tagy pre daný objekt. Nástroj Tagger už využíva väčšinu dostupných informácií pre tagovanie a nie je k dispozícii iný referenčný systém pre porovnanie. Môžeme teda aspoň porovnať tagy v databáze s menšími sadami vzoriek získanými z rozličných zdrojov. Chyby v tagoch vzoriek ale väčšinou poukazujú na chyby v externých nástrojoch.

Pre testovanie je použitá databáza s jednou replikou. Oba uzly bežia na serveroch DELL PowerEdge M630 s procesormi 2x Intel Xeon E5-2699 v3, 2.3GHz (hyperthreading, 72 vláken), 256 GB RAM, HDD 2x300 GB 15k RPM. Běží tam 64b Debian Stretch. Procesy na spracovanie prichádzajúcich udalostí bežia na rovnakom serveri, ktorý má ale len 128GB RAM.

7.1 Jednotkové a integračné testy

Pre základné testovanie implementácie jednotkovými a integračnými testmi sa používa testovací framework `unittest`, ktorý je súčasťou štandardnej knižnice v jazyku Python.

Na spúšťanie testov je použitý nástroj `green`[1]. Jeho výhodou je prehľadný farebne rozlíšený výstup, je postavený nad knižnicou `unittest`, poskytuje pokrytie kódu, je to open-source projekt a stále aktívne vyvíjaný.

Jednotkové testy testujú len základnú funkcionalitu kódu a nijako pritom neinteragujú s databázou či externými službami. Takáto funkcionalita buď nie je testovaná alebo je komunikácia s nimi fingovaná pomocou `unittest.mock` knižnice. Celkové pokrytie jednotkovými testmi zobrazuje tabuľka 7.1 a je to len 62%, čo je spôsobené tým, že veľká časť nástroja práve vyžaduje prácu s externými službami.

Integračné testy už naplno pracujú so všetkými službami rovnako ako produkčná verzia nástroja Tagger. Testujú teda reálnu funkcionalitu a interakciu nástroja. Ukážka z výstupu ich spustenia je v ukážke 7.2. Doba ich behu je výrazne vyššia, pretože sú spustené len

[1] https://github.com/CleanCut/green

```
Name                                                    Stmts       Miss       Cover
-------------------------------------------------------------------------------
tagger/av_detections.py                                   110          0        100%
tagger/clusty.py                                           35          0        100%
tagger/det_hits.py                                        134         87         35%
tagger/tag.py                                             295          0        100%
tagger/tagger_db.py                                       401        338         16%
tagger/time_delta.py                                       50          0        100%
tagger/yara_rules.py                                      152          0        100%
...
TOTAL                                                    2010        755         62%
Ran 237 tests in 0.572s using 72 processes
OK (passes=237)
```

Kód 7.1: Ukážka z výstupu spustenia jednotkových testov s celkovým pokrytím kódu

v jednom procese. Je to spôsobené prácou s reálnou databázou. Keďže jednotlivé testy testujú prípady ako počet vložených či aktualizovaných záznamov v databáze, nesmú byť spustené súčasne, aby si navzájom nemenili dáta. Možnosťou pre paralelný beh týchto testov je napríklad framework `testing.postgresql`[2]. Nevýhodu ale je, že je to len databáza reprezentovaná v pamäti, nie reálna databáza s ktorou pracuje produkčná verzia Tagger-u. Testovaním nad reálnou databázou máme väčšiu istotu, že aktuálna verzia bude pracovať správne aj po nasadení do produkcie.

Celkové pokrytie kódu integračnými testmi je 70%. Tieto testy už ale netestujú kód, ktorý nevyžaduje spoluprácu s externými nástrojmi a bol otestovaný jednotkovými testmi.

```
Name                                                    Stmts       Miss       Cover
-------------------------------------------------------------------------------
tagger/clusty.py                                           35         14         60%
tagger/det_hits.py                                        134          1         99%
tagger/det_hits_criteria.py                                21          0        100%
tagger/tagger\_db.py                                      401         30         93%
tagger/tasks/burger.py                                     76          1         99%
tagger/worker.py                                           44          0        100%
...
TOTAL                                                    1984        589         70%
Ran 135 tests in 40.602s using 1 process
OK (passes=135)
```

Kód 7.2: Ukážka z výstupu spustenia integračných testov s celkovým pokrytím kódu

Celkové pokrytie kódu jednotkovým aj integračnými testmi je zobrazené v tabuľke 7.3. Výsledok 90% je už dostatočne vysoký. Chýbajúce časti sú hlavne z kódu pre webové rozhranie a prepojenie s Burger databázou. V prípade webu je väčšina funkcionality testovaná

[2]https://pypi.org/project/testing.postgresql/

manuálne. Testovanie práce s Burger databázou by bolo časovo veľmi náročné, keďže Tagger požiadavky na Burger trvajú niekoľko minút, aj desiatky minút.

```
Name                                          Stmts      Miss      Cover
-----------------------------------------------------------------------
tagger/clusty.py                                 35         0       100%
tagger/det_hits.py                              134         0       100%
tagger/det_hits_criteria.py                      21         0       100%
tagger/tagger_db.py                             401        30        93%
tagger/tasks/burger.py                           76         1        99%
tagger/worker.py                                 44         0       100%
...
TOTAL                                          2086       209        90%
Ran 372 tests in 45.208s using 1 process
OK (passes=372)
```

Kód 7.3: Ukážka z výstupu spustenia jednotkových a integračných testov s celkovým pokrytím kódu

7.2 Rýchlosť API

K dátam uloženým v databáze pristupujú analytici pomocou webového rozhrania či API.

Ďalším konzumentom dát je samotný Tagger, ktorý ich používa pre štatistiky generované pomocou Burger služby. V tomto prípade ale nároky na rýchlosť databázy sú minimálne, pretože tieto požiadavky sú zadávané len manuálne. Teda určite nemôžeme ani hovoriť o niekoľkých požiadavkách za sekundu.

Táto časť sa zameriava teda len na testovanie rýchlosti a výkonu API. Za týmto účelom je vytvorený skript pre tzv. *stress* testy, ktorý vytvorí veľké množstvo vláken a každé z nich v danom časovom intervale neustále zasiela požiadavky na Tagger API. Po uplynutí časového limitu sa spočíta celkový počet úspešných a neúspešných odpovedí vo všetkých vláknach. Sledovanou hodnotou výkonu API je množstvo úspešne vybavených požiadaviek za jednotku času, teda napr. za sekundu.

Výstup skriptu je uvedený v prílohe A. Každý endpoint bol testovaný 60 sekúnd pomocou 200 vláken. Endpointy, ktoré chceli z databázy získať zoznam vzoriek alebo detekcií boli limitované len na zoznam dĺžky 1 000. Samozrejme, niektoré požiadavky môžu vyžadovať viac položiek, iné menej. Pre porovnateľnosť výsledkov testovania teda je stanovený jednotný počet požadovaných položiek.

Výsledky teda odrážajú výkon databázy a webového servera. Podľa dokumentácie použitého serveru `gunicorn` je odporúčaný počet procesov $4-12$. Tieto testy mali rovnaké výsledky pri použití minimálneho aj maximálneho odporúčaného počtu procesov. Teda webový server `gunicorn` zvláda veľký nápor požiadaviek dostatočne dobre a výkon je ovplyvnený hlavne databázou.

Z výsledkov môžeme vidieť, že API nemá problém odpovedať na okolo 2 800 až 3 000 požiadaviek za sekundu pre získanie tagu daného objektu. Získanie zoznamu 1 000 detekcií a detekčných definícii pre daný tag je okolo 2 400 až 2 800 za sekundu. Výrazne pomalšie je ale získanie zoznamu vzoriek pre danú rodinu, kedy je to len okolo 500 za sekundu.

Výsledky API testov sú zhrnuté v tabuľke 7.1.

Endpoint	**Počet odpovedí**
Tag pre súbor	2826
Tag pre detekciu	2703
Zoznam vzoriek danej rodiny	456
Zoznam 1000 Avast detekcií s daným tagom	2435

Tabuľka 7.1: Výsledky API testov ukazujúce počet vybavených odpovedí na jednotlivých endpointoch za 60 sekúnd.

Tieto hodnoty sú získané ako priemerná hodnota pri 60 sekundovej neustálej záťaži. V prípade spustenia stress testov len po dobu jednej sekundy dostávame asi o 40% požiadaviek za sekundu viac pre získavanie tagov pre objekty. Ide asi ale len o krátkodobé maximá.

Tieto hodnoty sú dostatočné pri aktuálnom používaní platformy Tagger. V prípade výrazného nárastu požiadaviek za limity aktuálnej architektúry je možné spustiť webový server na viacerých serveroch s vhodným vyvažovaním záťaže, napríklad pomocu NGINX[3] serveru. Zvýšenie priepustnosti dotazov do databázy je možné zlepšiť vyšším počtom replík.

7.3 Rýchlosť spracovávania externých udalostí

Príchod externej udalosti znamená vloženie nového tagu do databázy pre objekt (vzorka, detekcia, . . .) alebo porovnanie nového tagu s aktuálne uloženým a jeho prípadnou aktualizáciou. V prípade niektorých udalostí, prevažne z nástroja Clusty, je nutné získať dodatočné informácie o objekte. Rýchlosť spracovávania týchto udalostí teda nezávisí len na Tagger architektúre, ale je ovplyvnená aj sieťovou komunikáciou a rýchlosťou vybavovania požiadaviek v používaných nástrojoch.

Otestovanie celkového výkonu v reálnom nasadení teda môže prebiehať nasledujúcim spôsobom. Konzumenti externých udalostí nimi naplnia interné fronty. Po určitej dobe sa konzumenti zastavia a spustia sa procesy na spracovanie prijatých úloh. Porovnaním času naplnenia front a ich spracovania môžeme vidieť, aké rezervy má ešte spracovávanie úloh pri danom počte zapojených procesov. V prípade pomalšieho spracovávania udalostí, než je ich príchod môžeme zvyšovať počet procesov. Vďaka Celery to môžeme robiť v rámci jedného serveru, či môžeme zapojiť do spracovávania celý ďalší server. Samozrejme nedá sa to zvyšovať neustále. Pri určitom počte procesov narazíme na limit rýchlosti čítania/zápisu databázy.

V reálnom nasadení neprichádzajú externé udalosti s rovnomerným rozložením, takže je vhodné pri tomto spôsobe testovania zvoliť dlhší časový úsek pre naplnenie front.

Podľa uvedeného postupu bol pre testovanie zvolený časový interval 20 minút pre konzumovanie udalostí. Po jeho uplynutí sa vo frontách nachádzalo celkom 115 000 správ z jednotlivých služieb. Následne boli tieto správy spracované paralelne 60 procesmi za 12,5 minút. Pri takejto rýchlosti spracovávania by teda bolo možné za 20 minút spracovať asi 185 000 správ. Z toho vyplýva rezerva aktuálneho systému asi 70 000 udalostí, teda asi 40%.

Rovnaký postup bol zopakovaný pre 120 paralelne pracujúcich procesov, ale rýchlosť spracovania udalostí bola len mierne vyššia. Z toho vyplýva, že tieto hodnoty sú už ohraničené databázou, sieťovou komunikáciou a prístupom k externým službám. Najväčším prob-

[3] https://www.nginx.com/

lémom pri takomto vysokom počte procesov už bol nástroj Clusty. Odpovede jeho API trvali priemerne niekedy až v jednotkách minút.

7.4 Rýchlosť aktualizácie dát v databáze

Na rozdiel od predchádzajúcej sekcie je toto testovanie zamerané len na výkon procesov spracovávajúcich nové informácie a výkon databázy. Nie je teda ovplyvnené nijako čakaním na externé služby.

Pre tieto účely môžeme použiť sadu dát z nástroja Clusty, ktorá obsahuje vzorky aj detekcie. Z výsledkov môžeme vidieť, ako rýchlo dokáže Tagger spracovať dané množstvo informácií. Testovacia sada obsahuje 50 000 zhlukov vzoriek a ich detekcií. Graf 7.4 zobrazuje zastúpenie veľkostí zhlukov v testovacej sade podľa počtu vzoriek. Graf 7.5 zobrazuje zastúpenie počtu detekcií zhlukoch testovacej sady. Každý zhluk sa spracováva samostatne ako jeden celok. Teda všetky jeho vzorky a zhluky sa ukladajú/aktualizujú v rámci jedného procesu a jednej databázovej operácie, hromadne. Celkovo je v tých zhlukoch 30 miliónov vzoriek a 3 milióny detekcií.

Pri príchode novej informácie môžu nastať tri situácie. Objekt sa v databáze nenachádza a bude vložený, objekt sa nachádza v databáze a má lepší tag, teda nebude sa nič aktualizovať. V treťom prípade je objekt v databáze, ale má horší tag a dôjde k aktualizácii informácií. Tieto scenáre sú otestované v nasledujúcej časti. Testy boli spustené s použitím 60 paralelných procesov v Taggeri.

Testovanie nad prázdnou databázou

Testovanie bolo najprv spustené nad prázdnou databázou. 50 000 zhlukov bolo spracovaných za celkový čas 354 sekúnd (21 240 sekúnd spolu na 60 procesoch). Okrem toho bol sledovaný celkový čas databázových operácií. Všetky operácie potrebné pre vloženie dát spolu bežali 17 800 sekúnd. Teda väčšina času je strávená v databáze a porovnávanie a vyhodnocovanie tagov na aplikačnej úrovni a sieťová komunikácia tvoria 16% času pri prázdnej databáze. V tomto prípade sa dáta len vkladali. Čítanie aktuálneho tagu v databáze sa vykonávalo vždy, ale výsledok bol prázdny a žiadne dáta sa nemuseli prenášať po sieti ani porovnávať. Vždy sa vložil nový tag.

Testovanie nad naplnenou databázou bez aktualizácie tagov

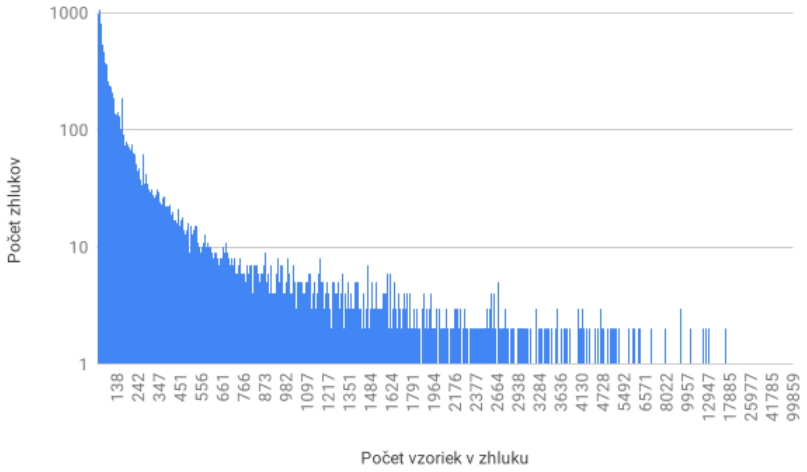
Ďalšie testovanie prebiehalo s rovnakými zhlukmi nad databázou, ktorá už nimi bola naplnená. Teda v tomto prípade sa neočakáva žiadna aktualizácia tagov. Pre všetky objekty Tagger vyhodnotí, že aktuálny tag je lepší. Je vlastne rovnaký a nevyžaduje si teda aktualizáciu. Dochádza teda len k získaniu aktuálneho tagu z databázy a jeho porovnaniu s novou informáciou. Bez následného zápisu.

Celkový čas spracovania 50 000 zhlukov bol 90 sekúnd (5 400 sekúnd spolu na 60 procesoch), z toho bolo 2 609 sekúnd, teda takmer polovica, strávených databázovými operáciami. Oproti predchádzajúcemu testu je to zvýšenie pomeru času databázových operácií a času strávenom v aplikačnej časti a sieťovej komunikácii. Je to spôsobené tým, že dochádzalo len k čítaniu z databázy a nič sa nezapisovalo.

Testovanie nad naplnenou databázou s aktualizáciou tagov

V tomto prípade databáza obsahovala objekty s tagmi, ktorým bol manuálne nastavený nízkoprioritný tag. Teda Tagger pre každý objekt rozhodol, že je nutné aktuálny tag aktualizovať s novým. Celkový čas spracovania 50 000 zhlukov v tomto prípade bol 344 sekúnd (20 640 sekúnd spolu na 60 procesoch). Z toho bol čas strávený v databáze bol 16 684 sekúnd.

Tieto výsledky sú podobné testovaniu nad prázdnou databázou. Rozdiel ale je, že v tomto prípade sa do databázy zapisovalo oveľa menej dát. Objekty sa tam už nachádzali a dochádzalo len k aktualizácii cudzích kľúčov v tabuľkách. Počet zápisov je teda výrazne nižší. Narástlo ale množstvo dát, ktoré sa museli načítavať z databázy pre vyhodnotenie a porovnanie tagov.



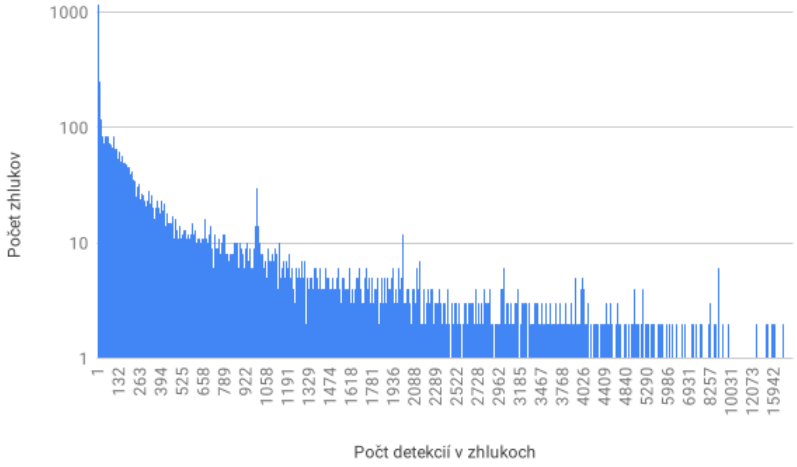
Obr. 7.4: Histogram počtu zhlukov s daným počtom vzoriek použitých pri testovaní výkonnosti databázy pre účely nástroja Tagger

Pri spustení so 120 procesmi bol výsledok vo všetkých troch testovaných scenároch len mierne lepší. Databáza je už naplno vyťažená. Možnosti zvýšenia jej výkonu boli popísané v sekcii 4.1.2.

Ako je uvedené na začiatku tejto sekcie, tieto testy vložili 30 miliónov vzoriek a 3 milióny detekcií za pár minút. Tento počet niekoľkonásobne prevyšuje denné množstvo novoprichádzajúcich vzoriek. V nástroji Clusty je priemerne do 1000 nových vzoriek za minútu, rovnomerne rozložených, čo nie je ani 1,5 milióna vzoriek denne [16]. Zdrojov informácií pre Tagger je niekoľko, ale ani po vynásobení počtu vzoriek v nástroji Clusty počtom zdrojov sa nedostaneme k 30 miliónom vloženým do 10 minút.

Samozrejme, pri vkladaní týchto 30 miliónov vzoriek boli všetky informácie priamo k dispozícii, nebola nutná komunikácia s inými nástrojmi. Externých správ spracovávaných nástrojom Tagger je samozrejme mierne viac, pretože mnohé z nich sú len aktualizácie už uložených informácií. Z testov je ale vidieť, že aktualizácie dát prípadne len porovnanie tagov bez aktualizácie sú ešte rýchlejšie než vkladanie nových dát.

Taktiež okrem vzoriek sa spracovávajú aj detekcie, ale tých je výrazne menej a stále sú výsledky testovania vysoko nad denným množstvom spracovávaných dát.



Obr. 7.5: Histogram počtu zhlukov s daným počtom detekcií použitých pri testovaní výkonnosti databázy pre účely nástroja Tagger

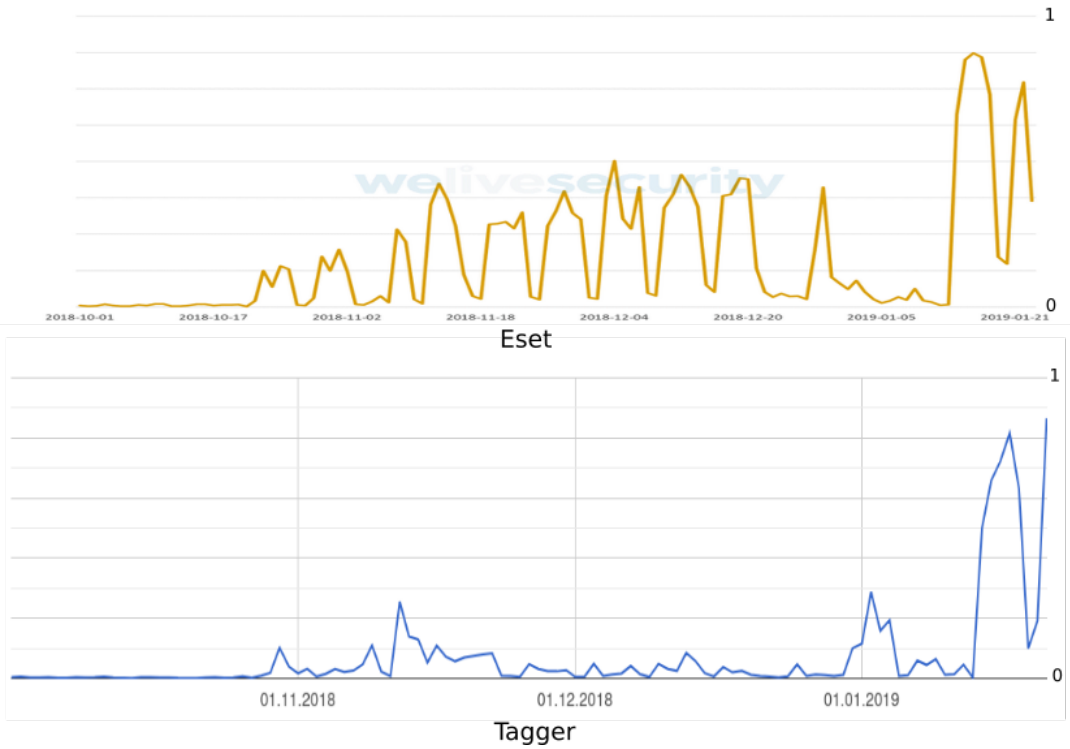
7.5 Správnosť tagov

Pre overenie správnosti prioritizovania tagov pre objekty môžeme použiť sadu manuálne klasifikovaných vzoriek, ktorá je k dispozícii v Avast Software. Z celej sady bolo vybraných 44 ,000 vzoriek, ktoré sa nachádzajú aj v Tagger databáze. Tieto vzorky neboli do Taggeru vložené manuálne. Dostali sa tam z niektorého z externých nástrojov. Z nich sa v 51 % prípadoch tag v Taggeri zhodoval presne s manuálnou klasifikáciou. V 24 % prípadoch bol typ v Taggeri trojan, pričom manuálne to bol iný typ. Táto nezhoda nie je ale veľmi kritická, keďže ako uvádza [27], typ trojan je veľmi generický. Mnohé škodlivé súbory vykazujú správanie viacerých typov a teda odlišnosti vo zvyšných 25 % vzoriek nemusia nutne znamenať výrazné chyby v tagoch.

Samotné prioritozovanie tagov rozhodovacím stromom je otestované kompletne sadou jednotkových testov. Nezhoda v tagu je teda pravdepodobne ovplyvnená chybou klasifikácie v externom nástroji. V takomto prípade nemá Tagger žiadnu možnosť jej odhalenia a táto chyba sa len spropaguje ďalej.

7.6 Generované štatistiky

Správnosť generovaných štatistík nie je jednoduché skontrolovať. Dobré porovnanie môže byť s verejne dostupnými výsledkami iných spoločností. Na obrázku 7.6 je porovnanie grafu vygenerovaného nástrojom Tagger a grafu od spoločnosti ESET. Ide o výskyt malvéru Troldesh (Shade) za obdobie štyroch mesiacov (október 2018 – január 2019). Grafy sú dosť podobné, hlavne v poslednom mesiaci. Prvé mesiace sú možno viac skreslené, ale dôvodom je ešte vtedy prebiehajúci vývoj nástroja Tagger. V tom čase mal Tagger ešte málo informácií v databáze.



Obr. 7.6: Porovnanie grafu výskytu denných útokov malvéru Troldesh od spoločnosti ESET (horný graf, prevzatý z [39]) a grafu z nástroja Tagger (dolný graf).

Kapitola 8

Záver

V tejto práci bol popísaný návrh implementácia a testovanie vysokovýkonného systému pre účely výskumu malvéru. Ide o platformu, ktorá agreguje, prioritizuje a vyhodnocuje informácie o klasifikácií vzoriek, detekčných definícií a ďalších objektov zaujímavých pri analýze malvéru. Zdrojom týchto klasifikácií, označovaných v rámci tejto platformy ako tagy, sú viaceré interné nástroje spoločnosti Avast Software. Tieto nástroje rôznymi prístupmi k analýze a klasifikovaniu produkujú veľa informácií, ktoré publikujú a ukladajú si u seba. Pre lepšiu podporu analýzy malvéru a vytvárania štatistík šírenia malvéru ich Tagger agreguje a vyhodnocuje, žiadne nové klasifikácie sám nevytvára.

Z týchto interných nástrojov bol predstavený hlavne nástroj pre zhlukovú analýzu súborov – Clusty, ktorý je primárnym zdrojom informácií. Pre jeho lepšiu integráciu s touto platformou boli nutné navrhnúť, implementovať a otestovať viaceré jeho vylepšenia či opravy týkajúce sa presnejších klasifikácií či publikovania správ.

Hlavným cieľom práce bolo navrhnúť platformu s dostatočným výkonom pre zvládnutie spracovania veľkého množstva denne prichádzajúcich informácií. Výkon sa týka implementácie spracovávania informácií aj databázy. S dôrazom na tieto požiadavky bola celá platforma implementovaná a otestovaná. Z testov sa ukázal jej dostatočný výkon pri aktuálnych požiadavkách, dokonca má ešte dostatočné rezervy. Výkon ale často je spomalený komunikáciou s externými službami a ovplyvniť to je mimo možnosti tejto práce. Naviac boli navrhnuté ďalšie možnosti zvýšenia výkonu platformy, ktoré nebolo zatiaľ nutné implementovať.

Medzi ďalšie možnosti rozširovania a vylepšovania platformy patrí napríklad možnosť pridávať ďalšie analytické dotazy, vylepšovať webové rozhranie pre užívateľov a rozširovať jeho funkcionalitu, automatizovane monitorovať prichádzajúce udalosti, API dotazy a výkon systému. Návrh platformy ďalej umožňuje jednoduché pridanie podpory pre tagovanie ďalších typov objektov – URL adresy, IoC, ... V prípade tagovania detekčných definícií je priestor na ich možné spresnenie, keďže niektoré môžu byť generické a detekovať viacero rodín malvéru. Taktiež obecne pre všetky objekty sa môže zaviesť podpora pre viacnásobný typ v tagu alebo viacnásobný tag. Dôvodom je to, že niektoré súbory vykazujú správanie viacerých malvérových typov.

Okrem testovania výkonu bola celá platforma otestovaná jednotkovými a integračnými testmi a nasadená do produkcie. Od nasadenia už obsahuje v databáze niekoľko desiatok miliónov detekcií a vyše stopäťdesiat miliónov súborov. Bolo pomocou nej vygenerovaných už množstvo štatistík pre analytikov.

Literatúra

[1] CARO: *A New Virus Naming Convention*. 1991, [Online; cit. 3. 12. 2018].
URL http://www.caro.org/articles/naming.html

[2] Chen, T. M.; Robert, J.-M.: *The Evolution of Viruses and Worms*. In *Statistical Methods in Computer Security*, 2004, ISBN 9780824759391.

[3] DataStax: *Introduction to Apache Cassandra*. [online, cit. 1. 3. 2019].
URL https://www.datastax.com/wp-content/uploads/2012/08/WP-IntrotoCassandra.pdf

[4] Dowd, K.; Severance, C.: *High performance computing*. Cambridge: O'Reilly, druhé vydanie, 1998, ISBN 1-56592-312-X, 446 s.

[5] Egele, M.; Scholte, T.; Kirda, E.; aj.: *A Survey on Automated Dynamic Malware Analysis Techniques and Tools*. In *ACM Computing Surveys (CSUR)*, ročník 44, ACM, 2012, s. 1 – 42, doi:10.1145/2089125.2089126.

[6] Eugster, P.; Felber, P.; Guerraoui, R.; aj.: *The many faces of publish/subscribe*. In *ACM Computing Surveys (CSUR)*, ročník 35, ACM, 2003, s. 114–131, doi:10.1145/857076.857078.

[7] Gessert, F.; Wingerath, W.; Friedrich, S.; aj.: *NoSQL database systems: a survey and decision guidance. Computer Science - Research and Development [online]*, ročník 32, č. 3, 2017, ISSN 1865-2034.

[8] Hager, G.; Wellein, G.: *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton: CRC Press, 2010, ISBN 978-1-4398-1192-4, 330 s.

[9] Kamil Kolonko: *Performance comparison of the most popular relational and non-relational database management systems*. 2018, [online, cit. 2. 4. 2019].

[10] Kaspersky Lab: *Kaspersky Security Bulletin 2018 Statistics*. [online, cit. 2. 1. 2019].
URL https://go.kaspersky.com/rs/802-IJN-240/images/KSB_statistics_2018_eng_final.pdf

[11] Khare, A.; Huang, Y.; Doan, H.; aj.: *A Fresh Graduate's Guide to Software Development Tools and Technologies*. 2012, [Online; cit. 8. 12. 2018].
URL https://www.comp.nus.edu.sg/~seer/book/2e/Ch06.%20Scalability.pdf

[12] Kovac, P.: *Fighting Malware With GPUs In Real Time*. [Online; cit. 25. 01. 2017].
URL http://on-demand.gputechconf.com/gtc/2015/presentation/S5612-Peter-Kovac.pdf

[13] Kshemkalyani, A. D.; Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008, ISBN 978-0-521-87634-6.

[14] Kuo, J.: *The Common Malware Enumeration (CME) initiative*. 2005, [Online; cit. 7. 12. 2018].
URL https://www.virusbulletin.com/virusbulletin/2005/09/common-malware-enumeration-cme-initiative

[15] Křoustek, J.: *Retargetable Analysis of Machine Code*. Dizertačná práca, Vysoké Učenie Technické v Brne. Fakulta Informačných Technoogií, 2014.

[16] Křoustek, J.; Zemek, P.: *Kategorizace malware vzorků*. Interná dokumentácia spoločnosti Avast Software.

[17] Leavitt, N.: *Will NoSQL Databases Live Up to Their Promise? Computer*, ročník 43, č. 2, 2010: s. 12 – 14, [online, cit. 3. 4. 2019].
URL http://www.cs.rochester.edu/courses/261/spring2017/termpaper/09/Unal_Yuchen_Poster.pdf

[18] Liu, K.; Tan, H. B. K.; Chen, X.: *Binary code analysis. Computer [online]*, ročník 46, č. 8, 2013, ISSN 0018-9162.

[19] Maksimov, D.: Performance Comparison of MongoDB and PostgreSQL with JSON types. 2015.

[20] Meine, S.: *Fundamentals of SQL Server 2012 Replication*. Simple Talk Publishing, 2013, ISBN 978-1-906434-98-4.

[21] Michael Bayer: *SQLAlchemy Documentation*. [online, cit. 27. 4. 2019].
URL https://docs.sqlalchemy.org/en/13/faq/performance.html

[22] Mohamed, M. A.; Altrafi, O. G.; Ismail, M. O.: Relational vs. NoSQL Databases: A Survey. *International Journal of Computer and Information Technology*, ročník 3, č. 3, 2014, ISSN 0018-9162.

[23] MongoDB, Inc: *Introduction to MongoDB*. [online, cit. 30. 12. 2018].
URL https://docs.mongodb.com/manual/introduction/

[24] Mullins, C.: *The Pros and Cons of Database Scaling Options*. 2017, [Online; cit. 8. 12. 2018].
URL
http://go.nuodb.com/rs/139-YPK-485/images/Database-Scaling-Options.pdf

[25] Niyizamwiyitira, C.; Lundberg, L.: *Performance evaluation of SQL and NoSQL database management systems in a cluster. International Journal of Database Management Systems*, ročník 9, č. 6, 2017: s. 1–24, ISSN 0975-5705, doi:10.5121/ijdms.2017.9601.

[26] Pang, G.: *Scalable Transactions for Scalable Distributed Database Systems*. 2015, [Online; cit. 11. 12. 2018].
URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-168.html

[27] Plaskoň, P.: *Automatická klasifikace shluků souborů*. Technická správa, Vysoké Učenie Technické v Brne. Fakulta Informačných Technoogií, 2018.

[28] Shahzad, R. M. K.: *Classification of Potentially Unwanted Programs Using Supervised Learning*. Blekinge Institute of Technology, 2013, ISBN 978-91-7295-247-8, 154 s.

[29] Sikorski, M.; Honig, A.: *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012, ISBN 978-1593272906, 766 s.

[30] Singh, N.; Khurmi, S. S.: *Malware Analysis, Clustering and Classification: A Literature Review*. In *International Journal of Computer Science and Technology*, ročník 6, 2015, ISSN 0976-8491.

[31] Solem, A.; et al: *Celery Documentation*, [online, cit. 19. 12. 2018].
URL https://media.readthedocs.org/pdf/celery/latest/celery.pdf

[32] Sterling, T.; Anderson, M.; Brodowicz, M.: *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann, 2017, ISBN 978-0-12-420158-3.

[33] Symantec: *Internet Security Threat Report*. ročník 23, 2018, [Online; cit. 1. 12. 2018].
URL https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf

[34] The PostgreSQL Global Development Group: *PostgreSQL 11.1 Documentation*. [online, cit. 19. 12. 2018].
URL https://www.postgresql.org/docs/manuals/

[35] Unal, G. S.; Zheng, Y.: *Distributed Databases: SQL vs. NoSQL*. 2017, [online, cit. 1. 4. 2019].
URL http://www.cs.rochester.edu/courses/261/spring2017/termpaper/09/Unal_Yuchen_Poster.pdf

[36] Veerman, G.; Breuk, R.: *Database Load Balancing*. 2012, [Online; cit. 9. 12. 2018].
URL https://www.os3.nl/_media/2011-2012/courses/lia/rory_breuk_gerrie_veerman_-_report.pdf

[37] Vineet, J.; Liu, X.: *A Survey of Distributed Message Broker Queues*. 2017, [Online; cit. 1. 12. 2018].
URL https://arxiv.org/pdf/1704.00411.pdf

[38] VirusTotal: *Welcome to YARA's documentation!*, [online]. rev. 14. 8. 2018, [cit. 18. 12. 2018].
URL https://yara.readthedocs.io/en/v3.8.1/

[39] Welivesecurity by ESET: *Russia hit by new wave of ransomware spam*. [online, cit. 1. 5. 2018].
URL https://www.welivesecurity.com/2019/01/28/russia-hit-new-wave-ransomware-spam/

[40] Ye, C.; Chiu, D.: *Peer-to-Peer Replication with Preferences*. [Online; cit. 10. 12. 2018].
URL https://staff.ie.cuhk.edu.hk/~dmchiu/infoscale.pdf

[41] Yishan, L.; Manoharan, S.: *A performance comparison of SQL and NoSQL databases.* In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM) [online]*, 2013, ISSN 1555-5798, s. 15 – 19.

Príloha A

Výstup stress testov pre API

```
$ ./support/api_stress_tests.py http://localhost/api --time-interval 60 --
    concurrency 200 --limit 1000
Tests will be run with the following parameters:
  URL: http://localhost/api
  Sample hash: 7927
      ae4890ec0a4db117cdf0a398672c0ea1fc341d5aed4029efbfe4ab980433
  Avast detection: Win32:Malware-gen|PE3-3D83C46000008D35F706D9F677F2CD9F|
      troj
  Avast det. definition: PE3-DEAF6E13000000363FBC2B18AC3F5183
  Strain: Sality
  Severity: malware
  Type: fileinfector
  Min. confidence: 99
  Interval: 60s
  Process count: 200
  Endpoints:
    http://localhost/api/sample_sha256/7927
        ae4890ec0a4db117cdf0a398672c0ea1fc341d5aed4029efbfe4ab980433/tag/
    http://localhost/api/avast_detection/Win32:Malware-gen|PE3-3
        D83C46000008D35F706D9F677F2CD9F|troj/tag/
    http://localhost/api/avast_detection_definition/PE3-
        DEAF6E13000000363FBC2B18AC3F5183/tag/
    http://localhost/api/strain/Sality/samples_sha256/?limit=1000
    http://localhost/api/tag/avast_detections/?severity=malware&type=
        fileinfector&strain=Sality&min_confidence=99&limit=1000
    http://localhost/api/tag/avast_detection_definitions/?severity=malware&
        type=fileinfector&strain=Sality&min_confidence=99&limit=1000

Stress testing http://localhost/api/sample_sha256/7927
    ae4890ec0a4db117cdf0a398672c0ea1fc341d5aed4029efbfe4ab980433/tag/
Processed 169570 requests and failed to process 0 requests in 60 seconds.
Average 2826.17 requests per second at 0.00% fail rate using 200 processes.
```

```
Stress testing http://localhost/api/avast_detection/Win32:Malware-gen|PE3-3
    D83C46000008D35F706D9F677F2CD9F|troj/tag/
Processed 162211 requests and failed to process 0 requests in 60 seconds.
Average 2703.52 requests per second at 0.00% fail rate using 200 processes.

Stress testing http://localhost/api/strain/Sality/samples_sha256/?limit
    =1000
Processed 27415 requests and failed to process 0 requests in 60 seconds.
Average 456.92 requests per second at 0.00% fail rate using 200 processes.

Stress testing http://localhost/api/tag/avast_detections/?severity=malware&
    type=fileinfector&strain=Sality&min_confidence=99&limit=1000
Processed 146137 requests and failed to process 0 requests in 60 seconds.
Average 2435.62 requests per second at 0.00% fail rate using 200 processes.
```