VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

OPTIMALIZACE SYSTÉMU SURICATA ZREDUKOVANÍM MEZIVLÁKNOVÝCH ZÁVISLOSTÍ

OPTIMIZING THE SURICATA SYSTEM BY REDUCING CROSS-THREAD DEPENDENCIES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ADAM KRÍŽ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. LUKÁŠ ŠIŠMIŠ**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

156389

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Kríž Adam**
Program:	Informační technologie
Název:	**Optimalizace systému Suricata zredukovaním mezivláknových závislostí**
Kategorie:	Počítačové sítě
Akademický rok:	2023/24

Zadání:

1. Podrobně se seznamte se systémem Suricata IDS/IPS a jeho vnitřní architekturou.
2. Nastudujte možnosti uložení dat do bezzámkových a případně jiných datových struktur.
3. Navrhněte nahrazení sdílených datových struktur vhodnějšími strukturami.
4. Návrh experimentálně implementujte a vyhodnoťte.
5. Diskutujte dosažené výsledky a možnosti dalšího pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce:	**Šišmiš Lukáš, Ing.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	30.10.2023

Abstrakt

V dobe internetu je pripojenie ku globálnej sieti možné už takmer z každého typu zariadenia. Pripojenie na internet umožňuje napríklad chladnička, vchodové dvere, inteligentné hodinky a ďalšie. S rastúcim počtom zariadení, ktoré vyžadujú pripojenie na internet sa do popredia dostáva bezpečnosť a ochrana súkromia užívateľa. Jedným z riešení, ktorým môže byť zabezpečená ochrana siete je Suricata, ktorá sa využíva na detekciu sieťových hrozieb a udalosti. Môže byť implementovaná ako monitorovací systém sieťovej prevádzky alebo ako aktívna ochrana a predchádzať potencionálnym hrozbám. Táto práca bude zameraná na optimalizáciu systému Suricata v režime IDS (detekcie hrozieb a udalosti). Výsledkom práce budú zmenené určité dátové štruktúry, ktoré zredukujú medzivláknové závislosti. Následkom bude predpokladaný nárast výkonu v podobe šetrenia procesorového času a zvýšenia objemu spracovávaných paketov súčasne. Dosiahnuté výsledky budú podrobne popísané a zhodnotené na konci bakalárskej práce.

Abstract

In the age of the internet, connection to the global network is possible from almost any type of device. Just to name a few: a fridge, a front door, a smart watches and more. With the growing number of devices that require an internet connection, the security and protection of the user's privacy comes to the foreground. One of the solutions that can be ensured network protection is Suricata, which is used to detect network threats and events. It can deploy as a monitoring system or it can be an active prevention system. Aim of this work will be optimizing the Suricata system in IDS mode (Intrusion Detection System). As a result of the work, certain data structures will be changed, which will reduce cross-thread dependencies. The result will be an expected increase in performance in the form of savings processor time and increasing the volume of processed packets at the same time. Achieved results will be described in detail and evaluated at the end of the bachelor's thesis.

Kľúčové slová

Suricata, optimalizácia, paket, Flow tabuľka, zámky, AF_PACKET, RTE_HASH, tok, DPDK, IDS

Keywords

Suricata, optimization, packet, Flow-table, locks, AF_PACKET, RTE_HASH, flow, DPDK, IDS

Citácia

KRÍŽ, Adam. *Optimalizace systému Suricata zredukovaním mezivláknových závislostí.* Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Lukáš Šišmiš

Optimalizace systému Suricata zredukovaním mezivláknových závislostí

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Lukáša Šišmiša. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Adam Kríž
3. mája 2024

Poďakovanie

Chcel by som veľmi poďakovať pánovi Ing. Lukášovi Šišmišovi za vedenie mojej práce. Veľmi som mu vďačný za jeho cenné a odborné rady a za jeho ľudský prístup. Počas písania tejto práce boli momenty kedy práca nenapredovala podľa očakávaní. V týchto momentoch sa mi dostávalo od vedúceho práce aj dostatočnej motivácie v práci pokračovať.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

S rastom digitálnej transformácie a stále narastajúcim počtom na sebe závislých komunikujúcich zariadení, sa počítačové siete stávajú kritickým komunikačným prostriedkom vo všetkých aspektoch nášho života. Vládne inštitúcie, veľké či malé podniky nevyhnutne vyžadujú trvalé pripojenie k internetu na svoju činnosť.

S extrémne rapídnou expanziou digitálnych sietí, ktorá je spojená s rapídnym vývojom moderných technológii, narastá počet hrozieb a rôznych bezpečnostných incidentov. Na základe toho sa kladie čoraz väčší dôraz na bezpečnosť, ochranu osobných údajov a informácii a dostupnosť služieb. Počítačoví zločinci, organizácie, dokonca aj štáty vyvíjajú čoraz sofistikovanejšie techniky na narušenie normálneho chodu siete, krádeže dát a monitorovania cudzích zariadení. Bezpečnosť počítačových sietí sa tak stáva jednou z najdôležitejších oblastí informatiky.

Dnes už poznáme rôzne typy útokov na počítačové siete, spôsoby ochrany či zníženia rizika napadnutia. Medzi známe počítačové útoky patria rôzne formy škodlivého softvéru: vírusy, trójske kone, spyware a iné. Tieto programy sa bez vedomia užívateľa dostanú na zariadenie obete, kde vykonávajú škodlivú činnosť ako napríklad spomalenie systému, mazanie, blokovanie alebo odcudzenie dát. Spôsobom, akým je možné sa vyhnúť týmto škodlivým programom je používanie antivírusového softvéru alebo systému Firewall. V rozsiahlych sieťach je populárnym bezpečnostným riešením nasadenie systému na detekciu anomálii v sieťovom toku (IDS - Intrusion Detection System, IPS - Intrusion Prevention System).

Obeťou útoku nemusí byť samotné zariadenie. Každým dňom sa hromadia prípady, kedy ľudom boli odcudzené citlivé údaje prostredníctvom falošných webových stránok alebo e-mailov. Útočníci sa vydávajú za dôveryhodné osoby alebo inštitúcie. Užívateľovi tak stačí len vyplniť formulár napríklad na falošnom webe banky, v domnienke, že sa jedná o originálny web a údaje mu boli odcudzené. V takýchto prípadoch je nutné, aby si každý užívateľ internetu kontroloval zdroje e-mailov, domény stránok a podobne.

Populárne útoky, voči ktorým sa je stále ťažké brániť a možná je len prevencia sú DoS útoky (Denial of Service - Odmietnutie služby). Útoky DoS sú zamerané na preťaženie cieľového systému alebo siete tak, aby prestali byť dostupné pre legitímnych používateľov. To sa dosahuje veľkým množstvom nelegitímneho sieťového dátového toku. Vhodnou prevenciu proti týmto útokom je rozloženie prevádzky medzi viaceré serveri, ktoré poskytujú služby. Cloud služby môžu pomôcť absorbovať veľký objem sieťového toku počas útoku. Existujú aj rôzne anti-ddos systémy (HAProxy, DataDome).

Pre túto prácu sú pre nás zaujímavé IDS (Intrusion Detection System) a IPS (Intrusion Prevention System) systémy. Tieto systémy slúžia k detekcii a ochrane siete pred neauto-

rizovaným prístupom, útokom alebo nezvyčajnou sieťovou prevádzkou. Nevýhodou týchto systémov je ich výkonnostná náročnosť na hardvérové prostriedky.

Hardvérový vývoj je časovo náročný kvôli rôznym obmedzeniam ako sú výrobné procesy, veľké finančné investície a testovanie. Softvérová optimalizácia môže byť efektívnym spôsobom, ako zvýšiť výkonnosť existujúcich systémov a zlepšiť ich efektivitu do príchodu novšieho hardvéru. Poskytuje rýchle a ekonomické riešenia vo vývoji. Náplňou práce, je konkrétne optimalizácia IDS a IPS systému Suricata. Pre optimalizáciu je nutné pochopiť na akom princípe funguje daný systém a hlbšie ho preštudovať.

Kapitola 2

Suricata

2.1 Úvod do IDS/IPS systémov

Začiatkom 21. storočia sa IDS systémy začali stávať najbežnejším bezpečnostným opatrením v budovaní ochrany siete. Nahrádzajú systémy Firewall, ktoré reagujú na základe čísla portu, protokolu alebo IP adresy. Firewally tak môžu rýchlo a účinne spracovávať sieťovú prevádzku, pretože nemajú žiadnu hĺbkovú kontrolu paketov. Ich nevýhodou je, nemožnosť detekcie v obsahu sieťovej prevádzky.

Začiatkom roku 2000 sa stali populárne nové útoky ako SQL Injections[1] a Cross site scripting (XSS)[2], ktoré priamo prešli cez Firewall. To je teda skutočný začiatok uvedenia IDS systémov do hromadnej prevádzky. Počas tejto doby IPS systémy využívalo ešte veľmi málo organizácii. Na popularite začali naberať až s ich postupným vývojom. Dôvodom nepoužívania boli obavy, že IPS systém by mohol blokovať anomálnu (nezvyčajnú) sieťovú prevádzku, ktorá by nemusela byť nebezpečná. Prevencia bola teda riešená tak, že anomálna aktivita bola do siete vpustená, IDS ju zachytí, upozorní organizáciu a následne sú vykonávané príslušné kroky aby sa hrozba zo siete odstránila.

2.1.1 Intrusion Detection Systems

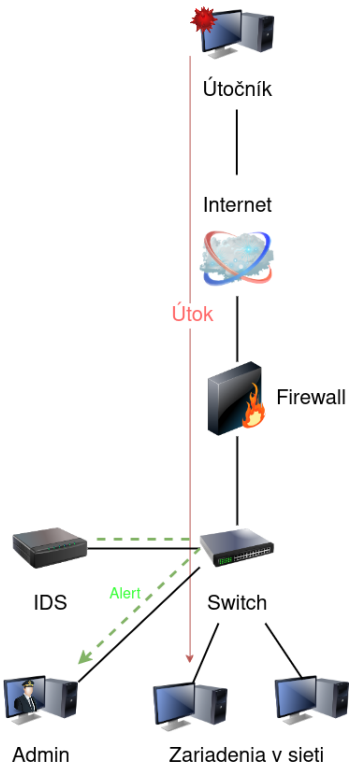
IDS[5] je systém navrhnutý na detekciu neautorizovanej alebo podozrivej prevádzky v sieti. Je to systém pasívnej ochrany. Jeho hlavnou úlohou je monitorovať sieťový tok a identifikovať známe a neznáme útoky na základe preddefinovaných pravidiel a vzorov. Následne generuje výstrahu, ktorá upozorní bezpečnostných pracovníkov, aby vyšetrili incident a vykonali nápravné opatrenia.

IDS môže detegovať rôzne typy útokov, vrátane pokusu o preniknutie, skenovanie siete, DoS a ďalšie. Možno ich klasifikovať niekoľkými spôsobmi. Jedným z nich je jeho umiestnenie v sieti. IDS môže byť nasadený na konkrétnom užívateľskom zariadení. To umožňuje monitorovať sieťovú prevádzku užívateľa, bežiace procesy, protokoly atď. Ďalej môže byť nasadený na úrovni siete, čo umožňuje identifikovať hrozby pre celú sieť. Voľba medzi užívateľským (hostiteľským) systémom detekcie narušenia bezpečnosti (HIDS) a sieťovým IDS (NIDS), je kompromisom medzi hĺbkou viditeľnosti a kontextom, ktorý systém prijíma.

Riešenia IDS možno klasifikovať aj podľa toho, ako identifikujú potencionálne hrozby. IDS založený na signatúrach používa na ich identifikáciu knižnicu signatúr známych hrozieb. IDS založený na anomáliách vytvára model "normálneho správania" chráneného systému a

[1] SQL Injections - https://www.imperva.com/learn/application-security/sql-injection-sqli
[2] XSS - https://www.synopsys.com/glossary/what-is-cross-site-scripting.html

hlási akékoľvek odchýlky. Hybridný systém využíva obe metódy na identifikáciu potenciálnych hrozieb.



Obr. 2.1: Schéma IDS systému v sieťovej topológii

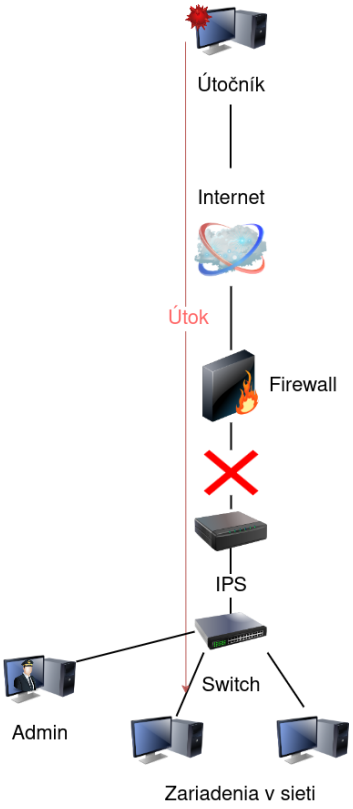
Na Obrázku 2.1 je možné vidieť sieťové zapojenie IDS systému. IDS je spustený na externom zariadení a skenuje celú sieťovú prevádzku. V prípade vzniku bezpečnostného incidentu, generuje výstrahu o incidente administrátorovi siete.

2.1.2 Intrusion Prevention Systems

IPS[6] je pokročilejšia verzia IDS. Na rozdiel od IDS, ktorého úlohou je sieť len monitorovať a detegovať útoky, IPS má navyše schopnosť aktívne reagovať a predchádzať nebezpečenstvu. Ide o aktívny systém ochrany.

Je schopný automaticky prijímať opatrenia pre zablokovanie alebo obmedzenie nebezpečného sieťového toku. Môže zahadzovať nežiadúce pakety, blokovať účastníkov komunikácie alebo resetovať spojenie, aby sa minimalizovali bezpečnostné riziká. Niektoré IPS systémy používajú tzv. "HoneyPot"[3], aby prilákali útočníkov, zaznamenali ich činnosť a následne im zabránili v dosiahnutí ich cieľov.

[3]HoneyPot - návnada vysokohodnotných dát



Obr. 2.2: Schéma IPS systému v sieťovej topológii

Na Obrázku 2.2 je znázornené zapojenie IPS v sieti. Nakoľko IPS musia byť schopné blokovať škodlivú aktivitu v reálnom čase, sú vždy umiestnené ”inline” v sieti. Sieťová prevádzka tak prechádza priamo cez IPS pred dosiahnutím cieľa. Najefektívnejšie umiestnenie je za Firewallom, keďže ten je vstupným a zároveň prvým ochranným bodom siete.

Systém Suricata je schopný pracovať v oboch režimoch. V tejto práci sa budeme venovať režimu IDS a to optimalizácii tohto systému. Cieľom je systém rozšíriť na menej výkonné stroje a zabezpečiť ochranu pre širšie skupiny užívateľov.

2.2 Úvod do Suricata

Suricata[7] je vysokovýkonný softvér na analýzu siete a detekciu hrozieb s otvoreným zdrojovým kódom, ktorý naberá na popularite v danej oblasti. Medzi partnerov, ktorí financujú vývoj systému patria napríklad Amazon Web Services, FireEye a Verizon. Je vyvíjaná neziskovou organizáciou Open Information Software Foundation (OISF) a keďže má otvorený zdrojový kód, do jej vývoja sa môže zapojiť každý a stať sa prispievateľom. Poprípade si ju môže každý upraviť podľa vlastných potrieb.

Medzi veľké výhody patrí jej izolovanosť v sieťovej topológii. To znamená, že funguje ako samostatný sieťový prvok, ktorý nenarúša správny chod iných ďalších bezpečnostných prvkov v sieti. Môžeme ju teda kombinovať s Firewallom, VPN a ďalšími technológiami. Ďalšou výhodou je prenositeľnosť. Suricata momentálne podporuje najpoužívanejšie platformy: Windows, Linux a MAC OS. Medzi hlavné nevýhody patrí hardvérová náročnosť. Pri vysokom počte spracovávaných paketov vo veľkých sieťach a pri použití veľa detekčných pravidiel ide o veľmi náročný systém z hľadiska využitia procesoru. Vyžaduje sa aj určité množstvo znalostí pre správne nastavenie konfigurácie a pravidiel pre zabezpečenie čo najefektívnejšieho chodu.

2.3 Porovnanie s inými systémami

Okrem systému Suricata existujú aj ďalšie systémy, ktoré sa zaoberajú analýzou sieťovej prevádzky a detekciou hrozieb. Ako najznámejší konkurent systému Suricata je Snort[2]. Ide tiež o IDS/IPS systém, rovnako ako Suricata. Na odhalenie potencionálnej hrozby používa sady pravidiel, s ktorými pakety porovnáva.

Snort bol predstavený v roku 1998 ako inovatívny nástroj v oblasti bezpečnosti siete. Vzhľadom na svoju technologickú vyspelosť a jedinečné vlastnosti si dlho udržiaval dominantnú pozíciu v oblasti kybernetickej bezpečnosti. V tom čase neexistovala žiadna konkurenčná aplikácia, ktorá by dokázala dosiahnuť porovnateľné výsledky v oblasti detekcie bezpečnostných hrozieb v sieti. S príchodom Suricaty do oblasti sieťovej bezpečnosti sa situácia zmenila. Suricata naberala na popularite, vďaka zlepšenej architektúre, ktorá umožňovala spracovávať pakety s využitím viacerých vlákien, čo predstavovalo vyššiu priepustnosť. Snort to zo začiatku nedokázal, a z hľadiska výkonu bola Suricata dlhodobo výkonnejšia. Firma Cisco v januári roku 2021 vydala Snort verziu 3, ktorá už taktiež podporuje viacvláknové spracovávanie paketov. V porovnaní výkonu oboch systémov, momentálne nie sú medzi nimi veľké rozdiely. Kedysi Snort podporoval menej aplikačných protokolov, respektíve bol menej flexibilný v narábaní s určitými typmi sieťovej prevádzky (napr. FTP - detekcia pravidiel v súboroch). Dnes už sú obidva systémy výkonovo veľmi podobné. Pri rozhodovaní, ktorý systém zvoliť ako najvhodnejší môže zavážiť skutočnosť, že Suricata má väčšiu komunitu prispievateľov na platforme Github. Z toho vyplýva, že v prípade určitého problému, alebo rýchlosti vývoja sa môže javiť lepšia ako Snort.

Ďalším systémom je Zeek[9]. Narozdiel od spomínaných systémov ide o monitorovací systém. Zeek generuje veľké množstvo záznamov (logs), ktoré popisujú aktuálny stav siete, rôzne druhy metadát, ktoré sú užitočné pre pochopenie, čo sa aktuálne deje na sieti. Zeek, môže byť skvelým doplnkom k Suricate.

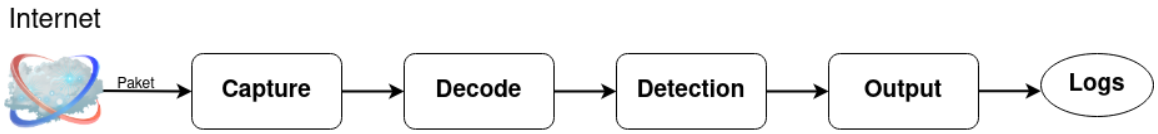
Môže sa stať, že IP pakety môžu byť sfalšované. IDS systémy ako Suricata alebo Snort, informácie z paketu len čítajú. Útočník môže v pakete nastaviť falošnú IP adresu a IDS systém to nedokáže zistiť. Táto skutočnosť môže spôsobovať generovanie falošne pozitívnych bezpečnostných incidentov IDS systémom. Je to častý jav, ktorý sa dá redukovať správnymi

nastaveniami, avšak nie na nulu. Týmito falošne pozitívnymi incidentami je nutné sa zaoberať, pretože by sa medzi nimi naozaj mohla byť skrytá hrozba. Vyžaduje si to individuálne dodatočné preskúmanie. V prípadnej kombinácii systému Suricata alebo Snort so Zeek, sa odhalenie týchto hrozieb stáva viac pravdepodobnejším.

2.4 Architektúra

2.4.1 Rozdelenie do modulov

Fungovanie systému Suricata je možné popísať ako rôzne celky, ktoré sú navzájom prepojené a každý má svoju úlohu. Rozdelenie do blokov je znázornené na Obrázku 2.3 aj s popisom jednotlivých blokov:



Obr. 2.3: Rozdelenie do blokov

1. `Zachytávanie paketov (Capture)`: Suricata začína zachytávať sieťové pakety. Môže zachytávať pakety z rôznych zdrojov, ako sú sieťové rozhrania (NIC) pomocou rôznych knižníc ako je napríklad AF_PACKET, DPDK, PF_RING a ďalšie. Pre testovacie a vývojové účely vie Suricata zachytávať pakety zo súborov PCAP[4] (Packet Capture). Tým je možné simulovať prípadnú nebezpečnú sieťovú prevádzku. Toto je počiatočný bod, v ktorom sa zbierajú sieťové pakety na analýzu.

2. `Dekódovanie paketov (Decode)`: Suricata dekóduje zachytené pakety, aby extrahovala rôzne vrstvy protokolov ako napríklad Ethernet, IP, TCP, UDP a aplikačné protokoly ako HTTP, SMTP a ďalšie. V prípade TCP spojenia sa taktiež rekonštruuje originálny sieťový tok, a to skladaním viacerých paketov dokopy. **Tokom sa rozumie postupnosť paketov medzi konkrétnym zdrojom a cieľom (identifikovaných IP adresou a číslom portu) s rovnakým protokolom**. Vďaka tomu je možné odhaliť skrytý útok, ktorého obsah je rozložený naprieč paketmi. Týmto dekódovaním môže Suricata skontrolovať obsah a metadáta každého paketu.

3. `Detekcia (Detection)`: Suricata používa súbor pravidiel alebo signatúr na detekciu potenciálne škodlivých alebo podozrivých aktivít v sieťovej prevádzke. Tieto pravidlá definujú špecifické vzory, správanie alebo charakteristiky spojené so známymi hrozbami alebo zraniteľnosťami. Po dekódovaní paketu je porovnaný s týmito pravidlami na identifikáciu zhôd. Ak paket spĺňa jedno alebo viac pravidiel, Suricata generuje výstrahu alebo záznam. Táto výstraha zvyčajne obsahuje informácie o zhodnom pravidle, zdrojových a cieľových IP adries, portoch a ďalších relevantných údajoch. Výstraha môže naznačovať potenciálne pokusy o intrúziu, komunikáciu s malvérom alebo iné bezpečnostné udalosti.

4. `Výstup (Output)`: Suricata poskytuje podrobné záznamy detekčných udalostí, ktoré možno použiť na ďalšiu analýzu, reakciu na incidenty a správu. Záznamy môžu byť

[4]PCAP - formátovaný súbor, ktorý ukladá zachytené pakety, obsahuje presnú kópiu každého bajtu paketu

v rôznych formátoch, ako sú JSON, EVE alebo vlastné formáty, čo umožňuje jednoduchú integráciu Suricaty s inými bezpečnostnými nástrojmi a systémami SIEM (Security Information and Event Management).

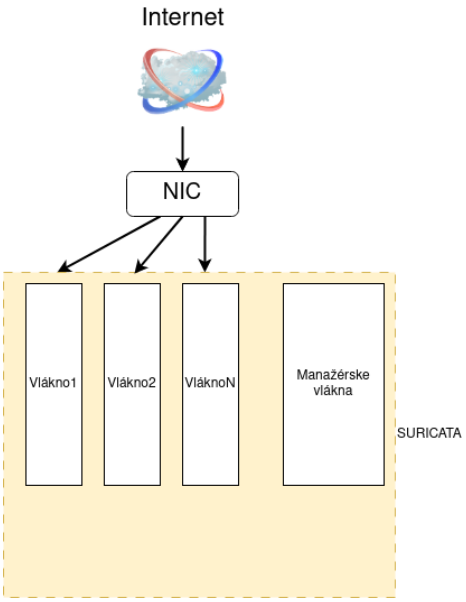
2.4.2 Režimy Behu

Spôsobom akým sú pakety spracovávané a porovnávané, nám udáva režim behu. Suricata umožňuje tri typy behu:

- `Workers` - Režim, ktorý podporuje viacvláknové spracovanie paketov. Každé vlákno sa stará o príjem paketov, ich dekódovanie, detekciu pravidiel a generovanie výstupu. Distribúciu paketov rieši sieťová karta, ktorá je zodpovedná za správne vyťaženie jednotlivých vlákien. Je nutné aby pakety patriace jednému sieťovému toku (rovnaké IP adresy a čísla portov) boli zasielané do rovnakého vlákna.
- `Autofp` - Tento režim sa používa hlavne pri IPS režime a pri spracovávaní PCAP súborov. Využíva sa jedno alebo viac vlákien pre zachytávanie a dekódovanie paketov. Pakety sú potom posielané do druhej skupiny vlákien, ktoré sa starajú o detekciu pravidiel. Medzi týmito skupinami vlákien je potrebná komunikácia, preto z hľadiska výkonu sa nejde o najlepšie riešenie.
- `Single` - Všetky pakety sú spracovávané iba jedným vláknom a nevyužíva sa paralelné spracovanie dát. Tento režim sa primárne používa na vývojové účely.

2.4.3 Workers

Pre túto prácu je najdôležitejší režim behu `Workers`, ako bolo vyššie spomenuté, je to režim behu s viacvláknovým spracovávaním paketov. Pojem `Worker` v systéme Suricata označuje konkrétne vlákno, ktoré vykonáva celý proces od spracovania až po vyhodnotenie paketov. Ide o paralelné spracovávanie paketov. Každé vlákno (`Worker`) je zodpovedné za porovnávanie sieťových paketov s definovanými pravidlami, ktoré špecifikujú vzory, správanie, charakteristiky spojené so známymi hrozbami alebo zraniteľnosťami.



Obr. 2.4: Rozdelenie do blokov

Ako je možné vidieť na Obrázku 2.4, vlákna pracujú nezávislé jedno od toho druhého, čo umožňuje systému Suricata efektívne škálovať a spracovávať veľké objemy sieťového toku. Prerozdeľovanie sieťového toku medzi viaceré vlákna prispieva k celkovej efektívnosti a Suricata tak môže analyzovať sieťové hrozby a reagovať na ne v reálnom čase, bez výrazného vplyvu na výkon (na primerane výkonnom stroji pri primeranej sieťovej prevádzke).

Je potrebné aby sa pri prerozdeľovaní sieťového toku, konkrétnemu vláknu pridelili vždy všetky pakety toho istého toku (pakety s rovnakým protokolom, zdrojovou a cieľovou adresou, zdrojovým a cieľovým portom). Ak by to tak nebolo, systém by nedokázal detegovať hrozby, pretože pri rozdelených paketoch by nemuselo byť možné nájsť potencionálne škodlivý obsah.

V konfiguračnom súbore je možnosť nastaviť počet vlákien, koľko daný užívateľ potrebuje. Limitovaný je len maximálnym výkonom zariadenia, na ktorom Suricata bude bežať. Stručne povedané, možnosť viac vláknového spracovávania je kľúčová vlastnosť, pre dosiahnutie vysokého výkonu.

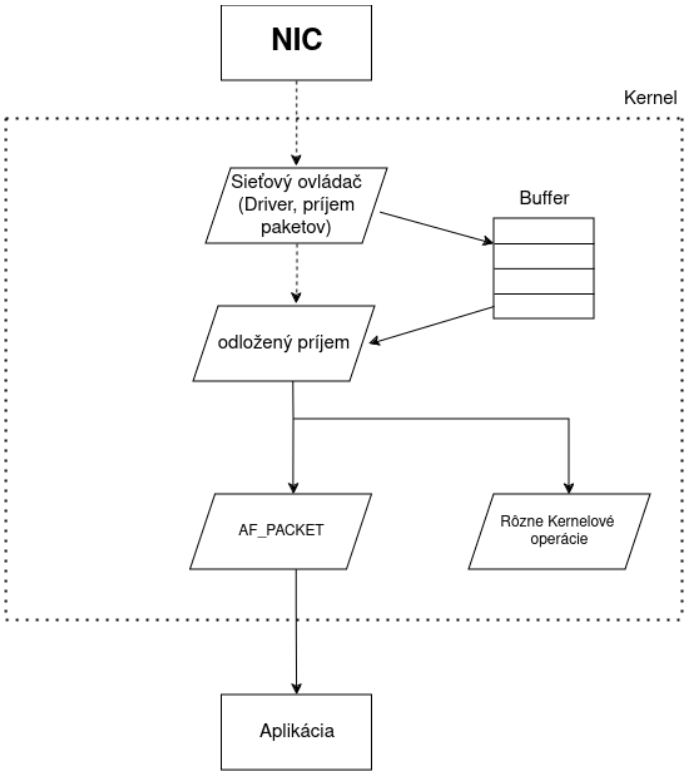
2.4.4 Možnosti zachytávania paketov

Ako bolo spomenuté v Sekcii 2.4.3, tak použitie režimu `Workers` je z hľadiska výkonu najoptimálnejšie. Výkon tohto režimu je možné upravovať výberom rozhrania určeného na zachytávanie paketov. Výber určitého rozhrania ovplyvňuje ako rýchlo budú zachytávané pakety zo sieťovej karty a preposielané samotnému systému. Suricata podporuje viac režimov zachytávania. Nižšie sú popísané režimy, ktoré sú pre túto prácu dôležité.

AF_PACKET

Sieťový socket[3] je koncový bod obojsmernej komunikácie medzi dvoma programami bežiacich v sieti. Je to prostriedok pre medziprocesovú komunikáciu, ktorý vytvorí pomenované kontaktné body, medzi ktorými prebieha komunikácia.

AF_PACKET[14] je typ Linuxového socketu používaného pre posielanie a prijímanie surových sieťových paketov priamo zo sieťového rozhrania. Surový paket[8] (raw packet) je termín používaný na označenie úplného paketu dát, ktorý je zachytený alebo odoslaný na sieťovom rozhraní bez akejkoľvek úpravy alebo spracovania na úrovni softvéru. To znamená, že surový paket obsahuje všetky údaje, ktoré boli poslané alebo prijaté na danej sieťovej karte, vrátane Ethernet hlavičiek a dát. Tento typ socketu je bežne používaný pri programovaní sieťových aplikácii.



Obr. 2.5: Fungovanie AF_PACKET

Mechanizmus získavania paketov pomocou rozhrania AF_PACKET znázorňuje Obrázok 2.5. Paket je na začiatku získaný sieťovým rozhraním a poslaný do kernelu (jadra operačného systému). Pred samotným hlbším spracovaním paketu je paket naklonovaný (z obrázku je možné vidieť, že určitý proces spracovania už prebehol). Naklonovaný paket je poslaný priamo do aplikácie, kde je možné s ním ďalej pracovať. Pôvodný paket je ďalej spracovávaný kernelom. Tento proces urýchľuje spracovávanie paketov, pretože práca so samotným paketom je vďaka naklonovaniu možná v skoršom čase. Proces klonovania môže spôsobiť malé straty výkonu.

V novších verziách Linuxového jadra bol pridaný kruhový buffer (mmbuf) medzi blok AF_PACKET a Aplikácia z Obrázku 2.5. Z tohoto bufferu je možné pristupovať priamo z aplikácie a rovnako aj z kernelu (t. j. prijímať a posielať pakety). Tým že sú pakety ukladané na rozmedzí aplikácie a kernelu, je zaistená redukcia určitého počtu systémových volaní. To prináša výkonnostný nárast.

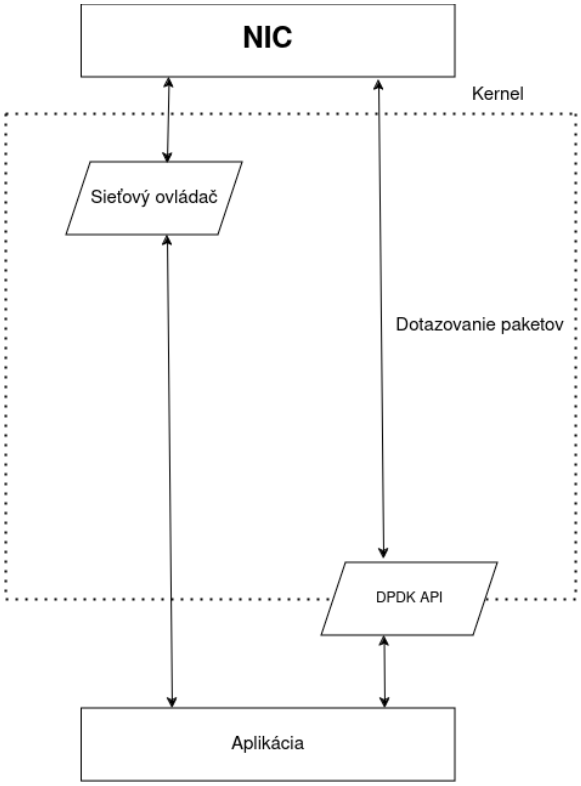
PCAP

Namiesto odoberania paketov zo sieťovej karty je možné ich čítať z PCAP súboru. Týmto spôsobom je možné simulovať reálnu prevádzku pre testovacie a vývojové účely. Prípadne otestovať správnu konfiguráciu systému alebo použitú sadu pravidiel.

DPDK

DPDK[10] (Data Plane Development Kit) je knižnica (framework) s otvoreným zdrojovým kódom, ktorá umožňuje vysoko rýchlostné spracovanie paketov. Pôvodne bol projekt vyvíjaný firmou Intel. V roku 2017 bol vývoj projektu presunutý pod Linux Foundation.

Knižnica je podporovaná viacerými architektúrami ako napríklad x86, ARM a PowerPC. Momentálne použitie DPDK je podporované operačnými systémami Linux a FreeBSD. Pre používanie DPDK knižnice je tiež nutné mať špeciálne ovládače pre sieťové karty, ktoré túto knižnicu podporujú. Tieto ovládače nie je možné používať na všetkých sieťových kartách[5].



Obr. 2.6: Fungovanie DPDK

Ako je možné vidieť na Obrázku 2.6 v porovnaní s režimom AF_PACKET z Obrázku 2.5 je minimalizovaná práca s kernelom. V kerneli beží sieťový ovládač, ktorý slúži len na konfiguráciu. Dotazovanie paketov je zabezpečené pomocou špeciálnych dotazovacích ovládačov (PMD[6]), ktoré prepájajú priamo aplikáciu so sieťovou kartou. Pakety teda obchádzajú kernel a je s nimi možné ihneď pracovať.

Dotazovací proces je pridelený ku konkrétnemu jadru na procesore. K dotazovaniu paketov dochádza neustále a tak nedochádza k prepínaniu kontextu na pridelenom jadre. Z

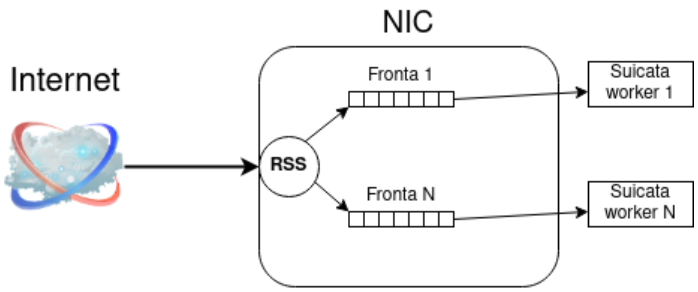
[5]Výrobcovia sieťových kariet podporujúcich DPDK - https://core.dpdk.org/supported/
[6]Poll Mode Driver - https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html

tohto dôvodu nie je možné, aby na danom jadre bežal ďalší proces a tak využitie daného jadra je 100%.

2.4.5 Receive Side Scalling (RSS)

RSS[1] je technológia používaná v moderných sieťových kartách (NIC). Slúži na zlepšenie efektivity spracovania prichádzajúcej sieťovej prevádzky, najmä v systémoch s viacjadrovými procesormi. Primárnym cieľom RSS je rozložiť pracovné zaťaženie spojené so spracovaním prichádzajúcich paketov medzi viaceré jadrá procesorov, čím sa zlepší celkový výkon siete.

V tradičnom nastavení sieťového spracovania sú všetky prichádzajúce pakety zvyčajne spracovávané jedným jadrom CPU. To sa môže stať prekážkou najmä v scenároch vysoko rýchlostných sietí, kde jedno jadro nemusí byť schopné držať krok s objemom prichádzajúcich paketov. Vďaka funkcii Receive Side Scaling je prichádzajúca sieťová prevádzka inteligentne distribuovaná medzi viaceré jadrá CPU. Každému jadru je priradená špecifická množina sieťových spojení alebo tokov. Táto distribúcia je často založená na hašovacích algoritmoch, ktoré berú do úvahy informácie z hlavičiek paketov, ako sú zdrojové a cieľové IP adresy alebo čísla portov.



Obr. 2.7: Využitie RSS v Suricate

RSS rozdeľuje sieťovú prevádzku do viacerých FIFO front. Pakety sú už potom predrozdelené pre viaceré vlákna. Každé vlákno už len odoberá pakety z danej fronty.

Túto technológiu vedia využiť len moderné sieťové karty, ktoré podporujú symetrické hašovanie pre prerozdeľovanie paketov. Problémom je, že daná technológia bola navrhnutá na bežnú sieťovú prevádzku, nie pre IDS systémy. Haš, podľa ktorého staršie sieťové karty prerozdeľujú pakety do front zvyčajne nie je symetrický. To znamená, že pakety jedného toku môžu skončiť v rôznych frontách. Ak sú pakety v rôznych frontách, poradie spracovávania paketov sa stáva nepredvídateľným. Ako príklad môže byť sieťový tok medzi klientom a serverom, kde do jednej fronty sieťovej karty bude pridelená prvá časť toku s TCP paketmi obsahujúcich "3-way handshake[7]". Druhej fronte budú pridelené pakety s dátami od serveru. Suricata si môže ako prvé zobrať pakety s dátami a keďže nespracovala najprv pakety obsahujúce "3-way handshake", označí tieto dáta ako neplatné. Žiadna z podporovaných metód zachytávania ako AF_PACKET alebo PF_RING nedokáže tento problém riešiť. Optimálnym riešením (v prípade nepodporovania asymetrického RSS) je teda zredukovať počet RSS front na jedna.

[7]TCP 3-way handshake - https://www.geeksforgeeks.org/tcp-3-way-handshake-process/

2.5 Konfigurácia súčasti Suricaty

Konfiguračný súbor **suricata.yaml** je kľúčovým komponentom na prispôsobenie správania sa systému Suricata na základe užívateľských požiadaviek, nastavenia siete a zabezpečenia. Formát YAML je dobre čitateľný pre človeka aj stroj, preto je konfigurácia veľmi intuitívna. Konfiguračný súbor obsahuje rozsiahle množstvo nastavení. Nastavovať je možné rôzne zložky, napríklad ukladanie rôznych výstupných logov, režimy behu, oprávnenia, prioritu pravidiel, režimy zachytávania paketov a ďalšie. K dôležitému nastaveniu určite patrí:

defrag (IP defragmentácia)

Niektoré pakety sa kvôli väčšej veľkosti fragmentujú (jeden paket sa rozdelí na viac). Záleží od rôznych nastavení konkrétnej siete. Aby Suricata bola schopná tieto pakety preskúmať správne, musia byť zrekonštruované do jedného paketu. To má za úlohu defragmentovacia jednotka. Zrekonštruovaný paket, je ďalej skúmaný zvyškom systému. V momente kedy Suricata príjme fragmentovaný paket, uloží si ho do pamäte, pre neskoršie použitie. Medzitým je schopná skúmať ďalšie pakety, kým nedorazí zvyšok fragmentovaného celku. Môže sa stať, že daný paket nikdy nedorazí, to by malo za následok, že daný paket ostane v pamäti. Z tohoto dôvodu sa nastavuje parameter `timeout` pre defragmentáciu. V praxi to znamená, že po uplynutí časového intervalu sa paket z pamäte uvoľní.

flow

Pri režime behu `Workers` bolo spomenuté, že pakety sú priraďované do skupín (tokov), na základe určitých spoločných vlastností (rovnaká zdrojová a cieľová IP adresa a ďalšie). Priraďovanie do týchto tokov prebieha interne. Na manažovanie všetkých tokov sa využíva pamäť. Čím viac tokov, tým viac využitej pamäte. S pamäťou súvisí viac nastavení. Jedným z nich je `memcap`, nastavuje sa tým maximálne množstvo bajtov, ktoré môžu byť využité na vytváranie tokov. Ďalším nastavením je `memcap-policy`, definuje správanie systému po vyčerpaní pamäte (napríklad vynechávanie nových paketov). K vyčerpaniu pamäte môže dochádzať nasledovným spôsobom. Pre každý paket nepatriaci do existujúceho toku sa vytvára nový tok. To by sa to dalo využiť v prospech útočníka tým, že by zahlcoval systém paketmi patriacimi do rôznych tokov. To by viedlo k vytvoreniu veľkého počtu tokov v tabuľke a následne by mohlo dojsť k rýchlemu zaplneniu tabuľky a preťaženiu systému. Suricata to rieši prechodom do núdzového režimu. Do núdzového režimu prechádza pri dosiahnutí určitého percenta obsadenosti tabuľky. Túto hodnotu je možné nastavovať. V núdzovom režime Suricata zníži časový interval expirácie tokov a tým sa pomaly začne uvoľňovať miesto pre nové toky. V prípade, ak to nepomáha ukončia sa niektoré toky, aj napriek tomu, že im nevypršal čas expirácie. Možné je nastavovať veľkosť hašovacej tabuľky, v ktorej sú jednotlivé toky uchovávané.

stream, reassembly

S monitorovaním tokov súvisí aj monitorovanie TCP spojení a rekonštrukcia správneho poradia paketov v danom spojení. V rámci TCP spojenia Suricata ukladá rôzne atribúty sieťového toku: stav, sekvenčné čísla a veľkosť okna. Veľkosť pamäte vyhradenú pre tieto informácie je možné nastavovať hodnotou `memcap` pod nastavením `stream`. Hodnotou `memcap` pod nastavením `reassembly` sa nastavuje veľkosť pamäte pre ukladanie dátových segmentov, aby ich bolo možné zrekonštruovať do pôvodného poradia.

host

Suricata obsahuje tabuľku, do ktorej sú ukladané IP adresy paketov pre neskoršiu prácu v rámci detekcie pravidiel. Veľkosť tabuľky je nastaviteľná hodnotou `hash-size`. Tabuľka je popísana podrobnejši v Podsekcii 3.1.3.

2.6 Pravidlá

Pravidlá systému Suricata (tiež aj signatúry) sú sady inštrukcií, podľa ktorých sa deteguje nezvyčajná alebo podozrivá aktivita v sieti. Pravidlá sú napísané v špecifickom jazyku, ktorý definuje kritéria pre identifikáciu konkrétnych typov sieťových udalostí. Detekčný modul využíva pravidlá na analýzu sieťovej prevádzky a na generovanie upozornení. Po vyhodnotení zhody s pravidlom je možné vykonať bezpečnostné opatrenia (generovať záznam, zahodiť paket ...). Príklad pravidla:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS
(msg:"HTTP Request to Known Malicious Site";
content:"GET /malicious-site"; sid:1000001;)
```

1. riadok špecifikuje, že pravidlo hľadá pakety s protokolom TCP, ktoré prichádzajú z akejkoľvek externej IP adresy, na ktorýkoľvek HTTP port na lokálnej sieti.
2. riadok obsahuje správu spojenú s pravidlom v čitateľnej podobe.
3. riadok je podmienka zhody obsahu, špecifikuje že obsah požiadavky HTTP musí obsahovať reťazec "GET /malicious-site", `sid` je identifikátor pravidla.

Pravidlá môžu zahŕňať rôzne možnosti reakcií na zhodu, ktoré je možné nastavovať do detailov. Detailnejší popis pravidiel sa nachádza v oficiálnej dokumentácii systému Suricata[8].

Kontrola pravidiel môže prebiehať rôznymi spôsobmi. Na začiatku sa vezme vzor z pravidla (vyhľadávaná sekvencia bajtov), ten môže byť určený napríklad kľúčovým slovom `fast_pattern`, ak nie je zadané kľúčové slovo, Suricata vyberie najsilnejší vzor (najdlhší, najviac variabilný). Vzory sa porovnajú s obsahom paketov. Porovnávajú sa iba tie pravidlá, ktorých vzory sa zhodovali s pravidlami. Tento systém sa nazýva Multi-Patern-Matcher. Pre tento typ vyhľadávania je možné nastaviť nastaviť rôzne algoritmy, predvolený algoritmus je Aho–Corasick (druh slovníkového algoritmu, ktorý hľadá vo vstupnom reťazci, prvky konečnej množiny reťazcov)[9].

Pre dosiahnutie vyššieho výkonu na výkonnejších strojoch je lepšie tento algoritmus zmeniť na Hyperscan[4], ktorý funguje na princípe deterministických konečných automatov. Je navrhnutý, aby efektívne porovnával obrovské počty (až desiatky tisíc) regulárnych výrazov naprieč dátami.

[8] rules - https://suricata.readthedocs.io/en/suricata-6.0.0/rules/intro.html
[9] Aho–Corasick - https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/

Kapitola 3

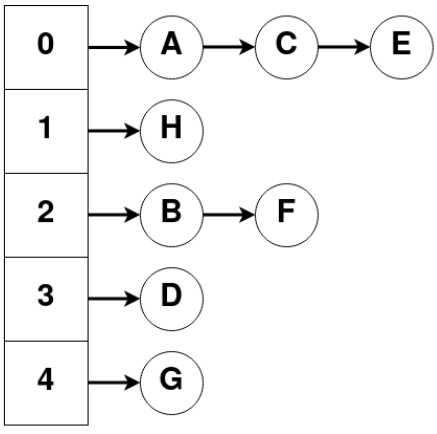
Návrh riešenia

3.1 Súčasný stav

3.1.1 Hašovacia tabuľka

Hašovacia tabuľka[12], alebo tiež tabuľka s rozptýlenými hodnotami je dátová štruktúra optimalizovaná pre potreby rýchleho vyhľadávania. Hašovacia tabuľka je N rozmerné pole a položkami poľa sú ukazovatele na lineárne zoznamy, do ktorých je možné uložiť položky, ktoré majú unikátny vyhľadávací kľúč. Pre adresáciu nám slúži hašovacia funkcia. Táto funkcia určuje, ako položky hľadať a ukladať, na základe vyhľadávacieho kľúča. Jej vstupom je vyhľadávací kľúč a výstupom je haš (charakteristika/otlačok vstupných dát). Pre rovnaké vstupné objekty je návratová hodnota (adresa) rovnaká. Nezaručuje, že pre dva rôzne objekty vráti odlišnú adresu. To spôsobuje, že veľkosť tabuľky nie je neobmedzená a tak môže nastať situácia, že odlišné objekty budu priradené na to isté miesto. Nazýva sa to kolízia alebo konflikt.

Ako bolo vyššie spomenuté, položky sú ukazovatele na lineárny zoznam a preto je možné v prípade kolízie, na jeden index pola umiestniť ľubovoľné množstvo položiek, tzv. synoným. Toto sa dá eliminovať použitím vhodnej hašovacej funkcie. V najhoršom možnom prípade by sa mohlo stať, že by všetky položky priraďovalo na jedno miesto.



[Klúč;Hodnota] : [0;A], [2;B], [0;C], [3;D], [0;E], [2;F], [4;G]

Obr. 3.1: Haš tabuľka

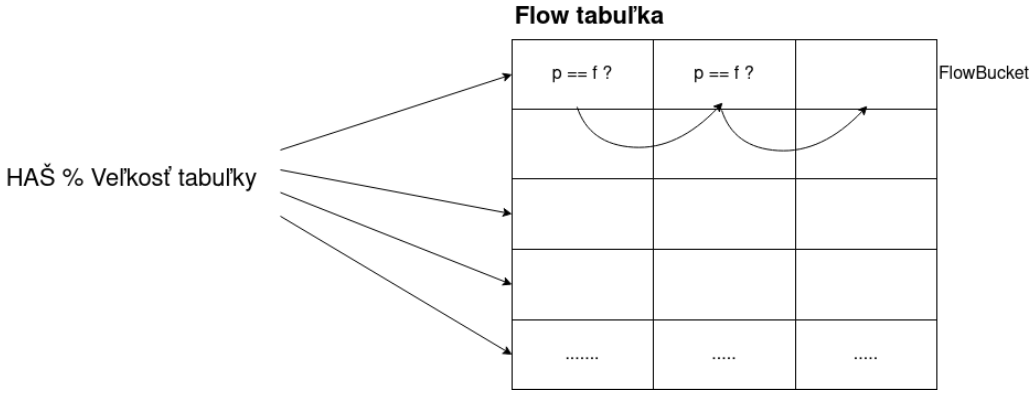
Vyhľadávanie prebieha analogicky k ukladaniu. Je možné vychádzať z Obrázku 3.1, ak by sa vyhľadávala hodnota E, hašovacia funkcia by nám vrátila nultý index a potom by sa postupne prechádzal zoznam a porovnával s každou položkou, pokým by sa nenašla. Týmto degraduje efektivita vyhľadávania kvôli sekvenčnému prechádzaniu zoznamu. Na riešenie kolízii existujú aj ďalšie varianty, ktoré sa v systéme Suricata nepoužívajú, ale nie sú pre túto prácu podstatné.

Použitie hašovacej tabuľky je efektívna metóda na vyhľadávanie pomocou kľúča, je nutné správne zvoliť hašovaciu funkciu, aby sa znižovala možnosť kolízie. K nevýhodám patria: pamäťová náročnosť, obmedzenie vyhľadávania len podľa celého kľúča. Nie je možné vyhľadávať na intervale (použitie aritmetických operátorov - väčší, menší) alebo pomocou podreťazca.

3.1.2 Flow tabuľka

`Flow tabuľka` je komplexnejšia implementácia hašovacej tabuľky, ktorá spĺňa špecifické potreby Suricaty. Tabuľka slúži na uchovávanie tokov reprezentovaných paketmi a aktualizáciu existujúcich. Počet riadkov tabuľky je daný nastavením v konfigurácii. Riadok v tabuľke sa nazýva `FlowBucket`. `FlowBucket` je zoznam tokov.

Spôsob používania tabuľky je nasledovný. Do systému príde paket a je pre neho vypočítaný haš, na základe jeho parametrov (zdrojová a cieľová IP adresa, číslo portu, protokol a ďalšie). Na základe hašu je pre paket pridelený `FlowBucket` (riadok tabuľky). Ďalej prebieha kontrola, či daný tok už existuje. Ak neexistuje, vytvorí sa. Ak áno, zoznam je prechádzaný tok po toku, dovtedy kým nie je nájdený zhodný tok a nahradíme ho novým paketom. Tento spôsob je znázornený na Obrázku 3.2.



Obr. 3.2: Flow tabuľka

Inicializácia a prístup do tabuľky je popísaný následovným kódom:

- **Inicializácia**:

```
flow_hash = SCMallocAligned(flow_config.hash_size *
sizeof(FlowBucket), CLS);
```

- **Prístup**:

```
FlowBucket *fb = &flow_hash[hash % flow_config.hash_size];
```

Prácu s tabuľkou vykonávajú manažérske, recyklačné a `Worker` vlákna. Počet manažérskych vlákien je možné nastavovať. Predvolene je použité jedno manažérske vlákno. `Worker` vlákna na základe prichádzajúcich paketov vytvárajú príslušne toky v tabuľke, alebo ich priraďujú už k existujúcim tokom. Ktoré, `Worker` vlákno bude spracovávať konkrétny tok, už bolo predstavené v Sekcii 2.1.

Z dôvodu šetrenia pamäte má každý tok vlastný časový interval života. Tieto intervaly sú nastavovateľné v rámci konfigurácie. Manažérske vlákna periodicky prechádzajú `Flow tabuľku`. Ako často a akú časť tabuľky (počet riadkov) bude manažérske vlákno prechádzať je dané funkciou `GetWorkUnitSizing`. Počet prechádzaných riadkov v jednom behu závisí od `MemCapPressure`[1]. Na základe počtu riadkov je určený čas spánku vlákna. V praxi to znamená že, ak práve Suricata nespracováva veľké množstvo prevádzky, tak kontroluje napríklad jednu šestinu tabuľky. Ak kontroluje práve malú časť tabuľky je predpoklad, že v najbližšej dobe by nemal nastať príchod väčšieho množstva paketov. Manažérske vlákno tak bude uspané na dlhší časový interval (maximálne jedna sekunda). Ak by bola situácia opačná, do systému prichádza väčšie množstvo prevádzky rôznych tokov a pamäť je postupne plnená, vtedy manažérske vlákno bude prechádzať väčšiu časť tabuľky (alebo celú časť). To mu zaberie dlhší čas a vlákno bude uspané na kratšiu dobu (minimálne 250 milisekúnd). Ak sa Suricata nachádza v núdzovom režime (Emergency mode[2]), daný výpočet sa nevykonáva a prechádza sa celá tabuľka v danom behu a čas spánku je 250 milisekúnd. V rámci prechádzania tabuľky manažérske vlákno kontroluje, či náhodou nevypršal toku jeho časový interval života. Ak áno, je tento tok odobratý z tabuľky a dočasne uložený do `aside_queue` (bočná fronta) a následne do `flow_recycle_q`, kde čaká na uvoľnenie z pamäte recyklačným vláknom.

Recyklačné vlákno je ďalší typ vlákna, ktorého počet je možné tiež nastavovať. Predvolene je použité jedno. Stará sa o uvoľňovanie tokov zo `flow_recycle_q`.

Pri odstraňovaní toku z `Flow tabuľky`, pristupovaní k `FlowBucket` alebo toku, je potrebná synchronizácia, aby nepristupovalo viac vlákien k jednému toku súčasne. To by mohlo spôsobiť nekonzistenciu dát. To znamená, keď vlákno pristupuje k `FlowBucket` alebo toku, uzamkne ho kým nedokončí prácu s ním. Ostatné vlákna vtedy musia čakať na odomknutie.

3.1.3 IPPair a Používateľská (Host) tabuľka

`IPPair a Používateľská tabuľka` sú ďalšie implementácie hašovacej tabuľky v Suricate. Oboje slúžia pre zrýchlenie, zefektívnenie a zjednodušenie práce s rôznymi udalosťami v rámci Suricaty. Implementačne ide o veľmi podobnú štruktúru ako `Flow tabuľka`.

`Používateľská tabuľka` je hašovacia tabuľka, ktorá má určitý počet riadkov. Riadok je reprezentovaný zoznamom (`HostHashRow`), ktorého prvkami je daný používateľ (dátová štruktúra, dôležitý atribút IP adresa). Haš, ktorý určuje riadok v danej tabuľke je vypočítavaný na základe IP adresy daného paketu. Pri vyhľadávaní používateľa v tabuľke je na začiatku vypočítaný haš, ktorý určuje riadok tabuľky. Následne je riadok prechádzaný

[1] V percentách vyjadrené vyčerpanie predalokovanej pamäte pre toky
[2] Prechod do tohto režimu je spôsobený vyčerpaním pamäte vyhradenej pre toky

sekvenčne užívateľ po užívateľovi a porovnávaný voči hľadanej IP adrese. Ak sa niektorý z používateľov zhoduje s porovnávanou IP, je daný používateľ v riadku presunutý na začiatok.

S `Používateľskou tabuľkou` môžu pracovať viaceré vlákna. Z tohoto dôvodu musí byť zabezpečená synchronizácia. Keďže dáta musia byť konzistentné, v jeden okamih môže zapisovať do tabuľky len jedno vlákno. To zabezpečuje zamykanie riadku tabuľky voči ostatných vláknam.

IP adresa používateľa sa zapíše do `Používateľskej tabuľky` použitím inštrukcie "`tag`" v rámci pravidla.

alert dns any any -> any any (dns.query; content:"evil";
tag:host,60,seconds,src; sid:1;)

V prípade použitia tohto pravidla Suricata vygeneruje upozornenie, pre každý paket v rámci DNS otázky, ktorý by v dátach obsahoval reťazec "evil". Atribút alebo kľúčové slovo `tag` znamená, že v prípade zhody s pravidlom v rámci sieťovej prevádzky, bude následujúca špecifická sieťová prevádzka (definovaná ďalšími atribútmi) označovaná.

Označovanie znamená, zaznamenávanie určitých paketov a ich výpis v rôznych logovacích súboroch. V tomto prípade je to označovanie všetkých paketov so zdrojovou IP adresou po dobu 60 sekúnd (v rámci sieťovej prevádzky, ktorá sa zhoduje s pravidlom).

`IPPair tabuľka` je z pohľadu implementácie úplne totožná s jediným rozdielom. Oproti `Používateľskej tabuľke` jej hlavným atribútom je dvojica IP adries. `IPPair tabuľka`, je využívaná napríklad pre ukladanie páru adries pri nastavovaní atribútu pravidla `Xbit`.

drop ssh any any -> $MYSERVER 22 (msg:"DROP BLACKLISTED"; xbits
:isset, badssh, track ip_src ;sid:4000000006;)

Toto pravidlo popisuje, že ak by v prípade sieťovej prevádzky nad protokolom SSH, jednoducho zahadzovalo pakety prichádzajúce z danej zdrojovej adresy (`ip_src`, poprípade zvoliť aj `ip_dst`). Atribút `ip_src` je možné nahradiť kľúčovým slovom `ip_pair`, v takom prípade pakety zahadzuje obojsmerne ale **iba pre daný tok**.

Z uvedeného vyplýva, že pre pravidlá používajúce `xbit` s atribútom `ip_dst, ip_src` je používaná `Používateľská tabuľka` pre ukladanie IP adresy. Pre `xbit` pravidla s atribútom `ip_pair` je používaná `IPPair tabuľka` pre ukladanie páru IP adries. Tieto tabuľky nám umožňujú podrobnejšie získavať ďalšie metadáta o konkrétnej sieťovej prevádzke. Získané metadáta slúžia pre analytické účely alebo pre špecifickejšie riadenie sieťového toku.

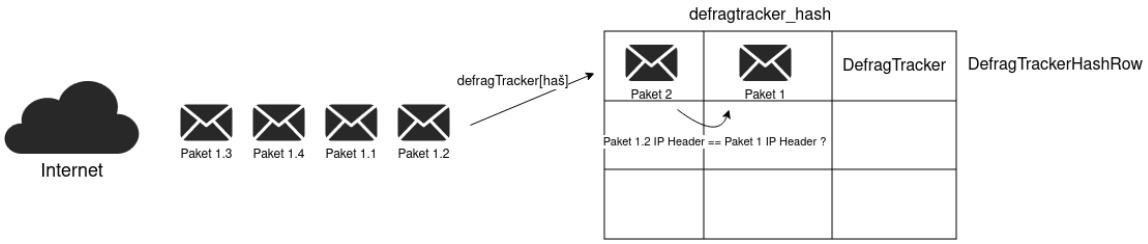
Všetky záznamy v oboch spomínaných tabuľkách majú taktiež časovú platnosť. O uvoľňovanie záznamov z týchto tabuliek sa stará tiež manažérske vlákno (recyklačné vlákno sa tohto procesu nezúčastňuje).

3.1.4 Defragmentačná tabuľka

`Defragmentačná tabuľka` (`defragtracker_hash`) je jednou z ďalších hašovacích tabuliek. Má za úlohu zaznamenávanie fragmentovaných paketov (fragmentovaný paket je označený konkrétnym bitom v hlavičke[3]), poprípade ich zoradenie do správneho poradia, ak boli pakety prijaté v rôznom poradí (poradie paketov sa tiež nachádza v hlavičke paketu). Na základe tohto procesu môžu byť pakety následne spracovávané ako jeden. Riadok tabuľky obsahuje pole dátového typu `DefragTracker`. `DefragTracker` reprezentuje paket ako celok. Obsah fragmentovaných paketov je priraďovaný na konkrétnu pozíciu poľa podľa toho, o

[3]IP fragmentácia - https://packetpushers.net/blog/ip-fragmentation-in-detail

ktorý paket ide. Ide o porovnávanie IP hlavičiek. V rámci priradenia prebieha prehadzovanie obsahu (rekonštrukcia obsahu paketov do pôvodného stavu) na správne miesto. Tento proces je znázornený nižšie na Obrázku 3.3.



Obr. 3.3: Proces defragmentácie paketu

3.1.5 Session pool a Segment pool

`Session pool` (`ssn_pool`) je dátová štruktúra pre zoskupenie prostriedkov pre vlákna, ktoré slúžia na uchovávanie aktívnych TCP spojení. `Session pool` je reprezentovaný poľom, ktorého prvkami sú ďalšie polia (`Pool *pool`). Každý jeden prvok patrí konkrétnemu vláknu. Pri vytvorení položky v `ssn_pool` je ako návratová hodnota identifikátor danej položky. Tento jedinečný identifikátor určuje pozíciu v poli a je priradený vláknu. Vďaka tomu každé vlákno pracuje s vlastným vyhradeným pamäťovým priestorom (pool).

Práca s `ssn_pool` prebieha následovným spôsobom. Spracovávaný TCP paket má priradený tok (`p ->flow`). Ďalej je skúmaný Suricatou či ide o paket, ktorý začína nové TCP spojenie alebo je to paket už z predchádzajúceho spojenia, ktoré je už uložené `ssn_pool`. Ak sa jedná o nové TCP spojenie, je funkciou `PoolThreadGetById` paketu priradený konkrétny `pool`.

- **Priradenie kontextu pre paket**:

```
p ->flow ->protoctx = PoolThreadGetById(ssn_pool, (uint16_t)id);
```

Vyššie uvedený `protoctx` je ukazovateľ na `void`. Je možné ho pretypovať na rôzne dátové typy. Ak je priradený `pool`, je tento kontext pre ďalšiu prácu pretypovaný na ukazovateľ dátového typu `TcpSession`.

- **Pretypovanie protoctx a vytvorenie premennej *ssn**:

```
TcpSession *ssn = (TcpSession *)p->flow->protoctx;
```

V rámci `ssn` sú ďalej skúmané a ukladané ďalšie potrebné parametre pre neskoršiu prácu s daným spojením (stav TCP spojenia, sekvenčné čísla, rôzne príznaky).

Opačná situácia je ak paket už patrí do existujúceho spojenia. Tým, že paket je súčasťou toku, tak pri už vytvorenom spojení na adrese kontextu je možné priamo pracovať so `ssn` (nutné pretypovanie `protoctx` a vytvorenie premennej `*ssn`). Odpadá nutnosť jeho priradenia ako bolo popísané vyššie.

`Segment pool` (`segment_thread_pool`) funguje fundamentálne na rovnakom princípe ako `ssn_pool` (pristupovanie k položkám a následná práca s nimi). Rozdiel je v práci na úrovni segmentov[4]. `Segment pool` slúži na zrekonštruovanie pôvodného poradia obsahu

[4]Segment - dátová jednotka na transportnej vrstve, paket bez IP hlavičky

paketov, ak boli pakety prijaté v inom poradí, prípadná kontrola správnosti poradia. Rekonštrukcia prebieha na základe sekvenčného čísla v rámci TCP hlavičky.

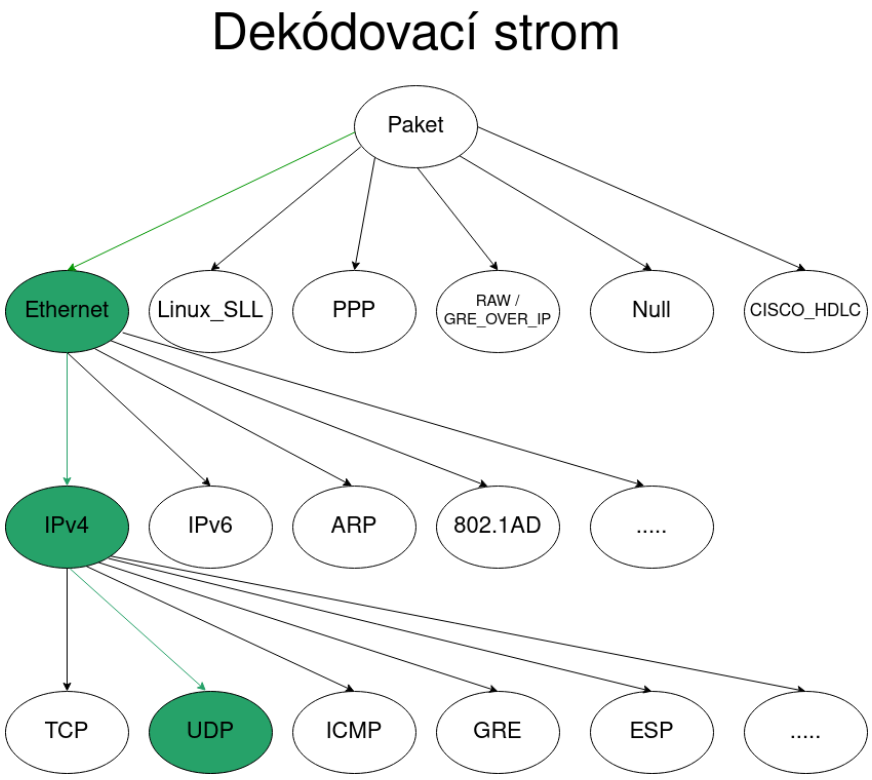
Vysvetlenie fungovania `Session pool` a `Segment pool` je veľmi zjednodušené. Ide o rozsiahlu a komplexnú funkcionalitu s veľkým počtom procesov. Pre túto prácu je uvedené vysvetlenie problematiky dostačujúce.

3.1.6 Cesta paketu

Na jednotlivých moduloch popísaných vyššie je možné demonštrovať príchod paketu (DNS paket, zariadenie s operačným systémom Linux) do systému a následne jeho životný cyklus. Demonštrácia je uvedená v režimu behu `Worker`. Celý proces je vykonávaný paralelne na viacerých vláknach.

Paket je prijatý sieťovou kartou v rámci sieťovej komunikácie a zachytený pomocou určitého rozhrania (AF_PACKET, DPDK, PF_RING alebo iné). Zachytený paket môže tiež pochádzať aj z PCAP súboru. V takom prípade by nešlo o režim `Worker`. V danom príklade je paket zachytený pomocou rozhrania AF_PACKET, následné je spracovávaný kernelom a poslaný do dekódovacieho modulu. V dekódovacom module je paket spracovávaný postupne od druhej vrstvy OSI modelu[5].

Fungovanie dekódovacieho modulu, je možné popísať dekódovacím stromov, ktorý je znázornený na Obrázku 3.4.



Obr. 3.4: Dekódovanie DNS UDP paketu

[5]OSI - referenčý model, ktorý popisuje fungovanie siete viz. https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi

Obrázok 3.4 znázorňuje dekódovací proces paketu. Paket je zachytený pomocou istého rozhrania, v tomto prípade AF_PACKET. Následne je dekódovaný protokol data-linkovej vrstvy OSI modelu. V danom pakete je dekódovaný Ethernet protokol, z obrázku je možné vidieť ďalšie podporované protokoly. Na základe zisteného protokolu sa volá príslušná dekódovacia funkcia (`decodeEthernet`). Volané funkcie vždy inkrementujú štatistiky, ktoré si uchovávajú rôzne informácie o počtoch, napríklad počet paketov, počet ethernet paketov a ďalšie. Rovnako vykonávajú kontrolu hlavičiek, či sú požadovanej veľkosti. V opačnom prípade Suricata vyhodnotí paket ako neplatný. Vnútornej štruktúre `_Packet`, ktorá reprezentuje reálny paket, sa nastaví príznak, že bola dekódovaná Ethernetová hlavička. Tento príznak je reprezentovaný uložením danej hlavičky do štruktúry `_Packet`.

Následne sa paket nachádza v ďalšej úrovni dekódovania, dekódovanie sieťovej vrstvy. Na sieťovej vrstve Suricata opätovne ponúka podporu rôznych protokolov, na obrázku sú vypísané len niektoré. V prípade DNS paketu ide o protokol IPv4. Pri dekódovaní IPv4 paketu dochádza k zvyšovaniu vnútorných počítadiel a následného nastavenia atribútov štruktúre `_Packet`. Konkrétne ide o hlavičku, zdrojová a cieľová IP adresa, dodatočné atribúty IPv4 paketu a protokol. Ak sa jedná o fragmentovaný paket, je mu nastavený príznak `PKT_IS_FRAGMENT`, a zaradený do rady pre neskoršiu defragmentáciu.

Následne je z paketu dekódovaný ďalší protokol vyššej vrstvy. Postup dekódovania ďalšieho protokolu je analogický k predchádzajúcemu. V tomto príklade ide o UDP paket. Sú mu nastavené príslušné hodnoty v štruktúre (hlavička, čísla portov, payload, protokol...) a na záver je pre daný paket vytvorený tok, prípadne je priradený k existujúcemu toku. K tomuto toku by boli ďalej priraďované ďalšie pakety, ktoré súvisia s daným tokom (rovnaká IP, protokol, číslo portu).

V rámci tokov, je paket opätovne preskúmavaný, či je to UDP alebo TCP paket. Pri TCP pakete, je pre paket vytvorené TCP spojenie, respektíve záznam v dátovej štruktúre pre zaznamenávanie TCP spojení (`ssn_pool`) a priradený kontext paketu. Priradenie kontextu prebieha spôsobom popísaným v Sekcii 3.1.5. V prípade ak paket patrí do existujúceho spojenia, kontext mu je priradený v rámci priradenia paketu do toku.

Pakety prijímané na sieťovej karte, ktoré sú súčasťou toho istého TCP spojenia nemusia byť prijaté v poradí akom boli odoslané. To znamená, že pakety sú poprehadzované. Tento stav je pre ďalšie správne fungovanie systému nežiadúci. V poprehadzovaných paketoch nie je možné vykonávať dostatočnú kontrolu nežiadúcich dát. Je nutné prijaté pakety poprehadzovať, aby boli v správnom poradí. Pre tento účel slúži špeciálna dátová štruktúra (`segment_thread_pool`) určená na rekonštrukciu pôvodného spojenia (poradia segmentov).

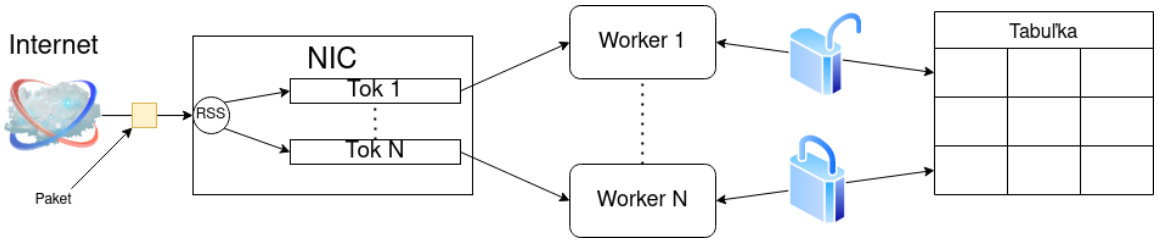
Rekonštrukcia TCP spojenia je vykonávané ihneď za vytvorením nového spojenia, prípadne vložením do existujúceho. Poradie segmentov určuje ich sekvenčné číslo, ktoré je súčasťou TCP paketu. V rámci tohto modulu prebiehajú aj ďalšie rôzne kontroly paketu spojené s transportnou vrstvou. Následne paket prechádza do detekčného modulu.

V detekčnom module sa opätovne kontrolujú vlastnosti paketu: použitý transportný protokol, čísla portov a IP adresy. Je to vykonávané z dôvodu optimalizácie. Pri UDP pakete sú TCP pakety vynechané nakoľko sa protokoly vzájomne vylučujú (platí to aj opačne). Týmto spôsobom je vytvorený určitý filter na základe, ktorého sú porovnávané pravidlá s paketmi. V prípade zhody paketu s určitým pravidlom, je paket priradený do fronty `alert_queue` pre neskoršie spracovanie. Na základe toho či bol nájdený podozrivý obsah alebo nie sa generujú výstupné záznamy (logs) o pakete a upozornenia o nájdení zhody s potencionálne nebezpečným obsahom.

3.2 Navrhované riešenie

V sekciách vyššie boli uvedené všetky dôležité abstraktné dátové štruktúry, ktoré sú zaujímavé pre túto prácu z hľadiska optimalizácie. Konkrétne ide o štruktúry: `Session pool`, `Segment pool` a `Flow`, `IPPair`, `Používateľskú`, `Defragmentačnú tabuľku` .

V rámci zadania je úlohou zrušiť medzivláknové závislosti naprieč dátovými štruktúrami, čo by malo viesť k optimalizácii Suricaty. Všetky zmienené štruktúry sú medzivláknové, to znamená, že k nim pristupuje každé vlákno.



Obr. 3.5: Práca viacerých vlákien - pôvodné riešenie

Spôsob, ako funguje pristupovanie k štruktúram je znázornené na Obrázku 3.5. Prístup je riešený formou zámkov, aby nedochádzalo k nekonzistencii dát, že by naraz zapisovali na to isté miesto dve vlákna. Ak chce vlákno pracovať s danou štruktúrou, skontroluje, či nie je zamknutá. Ak áno, čaká na odomknutie, ak nie môže pracovať s danou štruktúrou. `Tabuľka` na Obrázku 3.5 reprezentuje spomínané štruktúry (okrem `Session pool` a `Segment pool`, avšak z hľadiska pristupovania ide o totožný princíp).

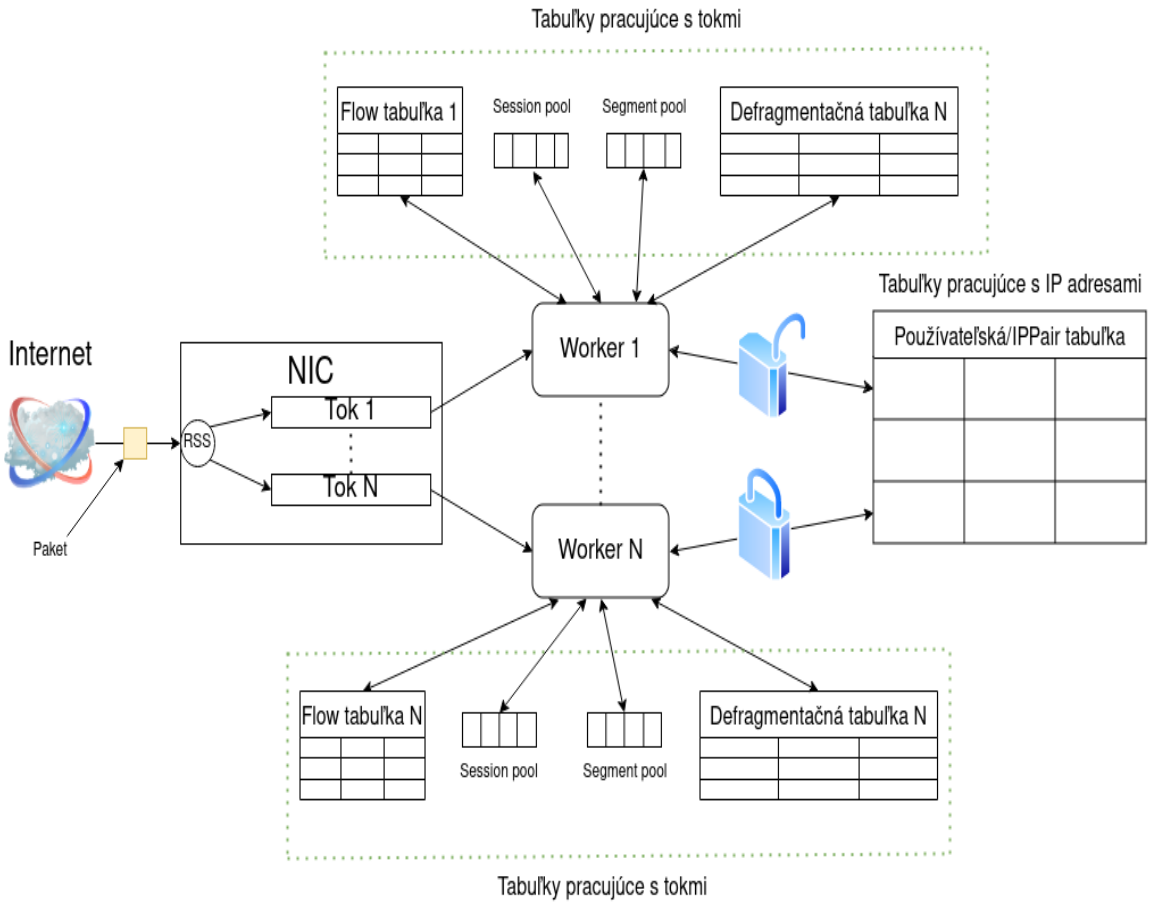
Proces zamykania, čakania a odomykania môže Suricatu v určitých momentoch spomaľovať. Navrhovaným riešením je dané štruktúry nevytvárať zdieľané ale vlastné pre každé vlákno. `Session pool`, `Segment pool` a `Flow tabuľka` pracujú len s tokmi. V Sekcii 2.4.5 bola spomínaná technológia RSS, ktorá už v rámci, sieťových kariet rozdeľuje prichádzajúce pakety do tokov. Ku každému vláknu (Worker), sú priraďované pakety toho istého toku. To znamená, že ak by malo každé vlákno vlastnú inštanciu dátovej štruktúry (Flow tabuľka / Session pool / Segment pool), do jej inštancie sa budú dostávať pakety tých istých tokov. Nenastane možnosť, že dve vlákna by mali pakety v štruktúre z toho istého toku.

Problémom je, že k niektorým tabuľkám musia pristupovať všetky vlákna, konkrétne ide o `IPPair tabuľku` a `Používateľskú tabuľku`, pretože pracujú s IP adresami, ktoré sú zdieľané medzi tokmi.

Napríklad, ak by komunikovalo určité zariadenie s konkrétnym serverom v rámci dvoch služieb s rozdielnym číslom portu. `Flow tabuľku` aj `IPPair tabuľku` by už malo každé vlákno vlastnú. Sieťový tok prvej služby by bol priradený jednej tabuľke a tok druhej služby priradený inej tabuľke. To že by každé vlákno spracovávalo toky v rámci svojej tabuľky, by mohlo priniesť zrýchlenie. V jednej z `IPPair tabuliek` je uložená dvojica IP adries, pretože IP adresy sú rovnaké v oboch tokoch. V prípade, že by pravidlá vyžadovali sledovanie ľubovolnej aktivity z IP adresy klienta alebo serveru, jednotlivé vlákna by nemali možnosť zdielať tieto informácie medzi sebou. Aby mohli vlákna pracovať správne bolo by nutné rovnaké dáta uložiť do každej tabuľky, ktorá by ich potrebovala. V takom prípade by šlo o duplicitné uloženie dát. Takýto stav je nežiadúci. Z tohoto dôvodu `IPPair tabuľka` a `Používateľská tabuľka` zostávajú zdieľanými štruktúrami.

`Defragmentačná tabuľka` pracuje len s IP adresami, nie s tokmi. Zaradíme ju do skupiny dátových štruktúr, ktoré pracujú s tokmi. Dôvodom je, že pracuje síce s IP adresami,

ale len na úrovni konkrétneho fragmentovaného paketu, ktorý je súčasťou určitého toku. To znamená, že nenastane situácia, v ktorej by fragmenty jedného paketu patrili do rôznych tokov. Z tohto dôvodu môžeme rozdeliť túto tabuľku na viac inštancií. Pre každé vlákno samostatnú.



Obr. 3.6: Práca viacerých vlákien - navrhované riešenie

Navrhované riešenie je znázornené na Obrázku 3.6. Na obrázku je znázornené, že každé vlákno bude mať vlastnú inštanciu `Flow tabuľky`, `Defragmentačnej tabuľky`, `Session pool` a `Segment pool`. Keďže vlákna budú mať každé vlastné inštancie pôvodne zdieľaných dátových štruktúr, manažérske vlákno a recyklačné vlákno nie je možné naďalej používať. Štruktúry nie je možné čistiť pôvodným spôsobom, pretože by bolo nutné zachovať synchronizáciu vlákien. Ich funkciu bude vykonávať samotné `Worker` vlákno v rámci svojich inštancií. Tabuľky, ktoré ostávajú zdieľané budú čistiť všetky vlákna.

`Worker` vlákno v rámci spracovávania paketov bude mať na starosti v určitých intervaloch čistenie všetkých tabuliek. Tu nastáva problém synchronizácie, že nesmie nastať moment, aby dve vlákna naraz čistili tú istú tabuľku. Aby sme tomu zabránili bude nutné pridať zámok. Ide o situáciu z Obrázku 3.5.

Tento prístup by mal priniesť určité výkonnostné zlepšenie len v určitých prípadoch. Ak sú pakety porovnávané s veľkým množstvom pravidiel, je predpoklad, že vlákna čakajú minimálne v miestach synchronizácie, pretože sú vyťažené spracovávaním sieťovej prevádzky v iných častiach systému.

Kapitola 4

Implementácia a testovanie

V Kapitole 3 boli abstraktne a funkcionálne popísané všetky dôležité dátové štruktúry, ktoré sú predmetom optimalizácie v rámci tejto práce. Predstavený bol návrh, akým smerom by sa mohla uberať implementácia navrhovanej zmeny systému Suricata. Kapitola je rozdelená na dve sekcie, ich názov vyplýva z pomenovania kapitoly.

Sekcia 4.1 detailne opisuje postupný proces implementácie a dodatočných informácii súvisiacich z daným procesom. Zahŕňa kroky od pridania súborov, zostavenia programu a komentáre určitých riadkov kódu. Implementácia prebiehala a taktiež je popísaná v poradí v akom sú popisované jednotlivé dátové štruktúry v Kapitole 3.

Sekcia 4.2 popisuje, akým spôsobom prebiehalo testovanie implementovaného riešenia. Popisuje prostredie, v ktorom boli testy vykonávané, aké prostriedky boli využité a aký dopad na systém mali vykonané zmeny. Súčasťou sekcie sú aj výsledky testovaní v podobe grafov a tabuliek.

4.1 Implementácia

4.1.1 Pridávanie súborov a zostavenie programu

Kód systému Suricata je napísaný v rôznych programovacích jazykoch. Na základe toho sú zdrojové súbory rozdelené do rôznych priečinkov (`python/, lua/ , rust/, ...`). Daná iplementácia bola napísaná v jazyku C. Všetky zdrojové súbory jazyka C sa nachádzajú v priečinku `src/`. Názvy súborov vychádzajú z architektonické návrhu, vďaka tomu je možné sa ľahko orientovať v kóde (`decode.c`, `flow-hash.c`, `ippair.c`, ...).

Implementácia pozostáva z úprav existujúcich častí kódu a vytvorenia ďalších súborov. Jednotlivé úpravy konkrétnych súborov sú popísané neskôr, pri popisoch implementácie daných dátových štruktúr vyplývajúcich z návrhu riešenia. Pridané boli následujúce súbory:

- `flow-hash-worker.{c,h}`
- `flow-worker-manager.{c,h}`

Samotným vytvorením a pridaním súborov do správneho priečinku, nie je zabezpečený preklad týchto súborov. Na preloženie Suricaty slúži systém MakeFile. Samotné súbory Makefile sú generované pomocou šablóny `Makefile.am`. Tieto šablóny generuje nástroj GNU Automake[1]. Aby bolo možné preložiť novo pridané súbory, je nutné zahrnúť ich do tejto

[1]GNU Automake - https://www.gnu.org/software/automake/

šablóny. Tento proces je potrebné vykonať pre každý súbor s príponou `.c` a `.h`. Pridanie súborov popisuje Výpis 4.1:

```
suricata\_SOURCES =
...
 flow-hash-worker.c flow-hash-worker.h \
...
 flow-worker-manger.h flow-worker-manger.h \
...
```

Výpis 4.1: Pridanie súborov do šablóny `Makefile.am`

4.1.2 Implementácia Flow tabuľky - pôvodná hašovacia tabuľka

Hlavná vetva systému Suricata sa nachádza v súbore `suricata.c`. V tomto súbore sa nachádza hlavný cyklus programu, v ktorom sú volané všetky ďalšie funkcie súvisiace s chodom systému. Nachádzajú sa tu tiež volania všetkých inicializačných funkcií, ktoré načítavajú dáta z konfiguračného súboru a inicializujú všetky dátové štruktúry potrebné pre správne fungovanie systému, vrátane tých, ktoré sú predmetom implementovaného riešenia.

Funkcia `FlowInitConfig` obsahuje nastavenie parametrov pre prácu s tokmi, ktoré sú nastaviteľné v rámci konfiguračného súboru (napríklad veľkosť haš tabuľky, počet predalokovaných tokov, chovanie systému po vyčerpaní pamäte a ďalšie). Ide o nastavenie globálnych vlastností, ktoré nie je potreba spracovávať na viacerých vláknach.

Ďalej obsahuje deklaráciu a inicializáciu dátovej štruktúry `flow_hash`, ktorá reprezentuje spomínanú `Flow tabuľku`. Táto štruktúra bola odstránená aj s jej kompletnou inicializáciou. Štruktúra `flow_hash` je dátového typu `FlowBucket`. V definícii štruktúry je jedným z atribútov tejto štruktúry zámok `SCMutex m`. Ten bol odstránený, kedže pri pristupovaní len jedným vláknom nie je potrebný.

Novo vytvorený súbor `flow-hash-worker.c` obsahuje novú deklaráciu `Flow tabuľky` a jej inicializačnú a deinicializačnú funkciu, tento krát už bez zámkov. Tieto kroky popisuje Výpis 4.2.

```
...
 thread_local FlowBucket *thread_flow_hash;
...
 void FlowHashThreadInit(void) {
thread_flow_hash = SCMallocAligned(flow_config.hash_size *
    sizeof(FlowBucket), CLS);
...
```

Výpis 4.2: Inicializácia `thread_flow_hash`

Nová deklarácia `Flow tabuľky` je doplnená o kľúčové slovo `thread_local`[2], ktoré zabezpečuje, že daná premenná bude zadeklarovaná lokálne pre každé vlákno.

Súčasťou funkcie `FlowInitConfig` je inicializácia štruktúry `flow_spare_pool`, ktorá plní funkciu úložiska pre predalokované toky (veľkosť je definovaná v konfiguračnom súbore). To znamená, že ak bol vytvorený nový tok, odoberie si voľné miesto z tejto štruktúry.

[2]thread_local - https://www.gnu.org/software/libc/manual/html_node/ISO-C-Thread_002dlocal-Storage.html

Aby bola zachovaná bezzámková charakteristika `Flow tabuľky`, bolo nutné túto dátovú štruktúru znova zadeklarovať ako `thread_local`. Rovnako boli odstránené zámky z definície štruktúry, pretože boli pre ďalšiu prácu s štruktúrou zbytočné.

Pre správne fungovanie bolo potrebné mierne upraviť všetky funkcie, ktoré používajú spomínané štruktúry na svoju činnosť a následne odstrániť prácu so zámkami. Zmeny sa týkajú následujúcich súborov:

```
flow_hash.{c,h}, flow.c, flow-spare-pool.c, flow-private.c,
flow-timeout.c, flow-worker.{c,h}
```

Vykonané zmeny spôsobujú že inicializačné parametre, ako napríklad veľkosť haš tabuľky sú teraz parametrami pre jednu konkrétnu inštanciu (nie je možné definovať rôzne rozmery pre rôzne tabuľky, analogicky to platí aj pre ďalšie parametre). Táto skutočnosť je zdokumentovaná formou komentára v konfiguračnom súbore.

4.1.3 Zlúčenie manažérskeho a recyklačného vlákna

V Kapitole 3 je popisovaná nekompatibilita manažérskeho a recyklačného vlákna s navrhovaným riešením. Je to zapríčinené architektúrou riešenia a aj samotnou realizáciou. Manažérske a recyklačné vlákna sú plnohodnotné vlákna. V deklarácii `Flow tabuľky` bolo zahrnuté kľúčové slovo `thread_local`, to znamená, že prístup k danej dátovej štruktúre má len samotné `Worker` vlákno. Navrhované riešenie pozostáva z nahradenia funkcie týchto dvoch vlákien `Worker` vláknom.

Každé vlákno v Suricate obsahuje určité dátové štruktúry, ktoré obsahujú rôzne parametre, atribúty vlákna a ukazovatele na funkcie. Tieto štruktúry sa líšia podľa toho na aký účel je vlákno používané. Základnou štruktúrou, ktorú obsahuje každé vlákno je `Threadvars`. Ako vyplýva z názvu ide o základné atribúty, ktorými je popísané každé vlákno (meno, id, typ vlákna, ukazovateľ na iné vlákna a ďalšie). Ďalšia štruktúra, ktorú vlákno obsahuje je už špecifická podľa úlohy akú samotné vlákno vykonáva. V tomto prípade by išlo o tieto typy štruktúr:

1. `Worker` - `FlowWorkerThreadData`
2. `Manager` - `FlowManagerThreadData`
3. `Recycler` - `FlowRecyclerThreadData`

Aby bolo možné zlúčiť vlákna do jedného a využiť pri tom pôvodné funkcie, s ktorými vlákna pracujú, je nutné aby tieto štruktúry respektíve ich položky boli zachované. Pre čo najjednoduchšiu prácu a maximálne využitie pôvodných funkcii, boli tieto pôvodné štruktúry zadefinované v rámci samotného `Worker` vlákna, čiže priamo štruktúry `FlowWorkerThreadData`. Implementáciu popisuje Výpis 4.3.

```
typedef struct FlowWorkerThreadData_ {
   ...
    FlowManagerThreadData *manager;
    FlowRecyclerThreadData *recycler;
   ...
   }
```

Výpis 4.3: Predefinovanie `FlowWorkerThreadData`

Súčasťou tejto redefinície je pomocná štruktúra, ktorá je využívaná pôvodným manažérskym vláknom `FlowTimeoutCounters`.

Hlavné funkcie, ktoré vykonávajú činnosť manažérskeho a recyklačného vlákna bežia v nekonečnom cykle. V cykle sú využívané pomocné premenné ako napríklad počet kontrolovaných riadkov `Flow tabuľky`, čas spánku vlákna, aktuálny čas programu a ďalšie. V implementovanom riešení už dané funkcie nebežia v nekonečnom cykle, ale sú volané sekvenčne vo funkcii `FlowWorker`. Funkcia je volaná nad každým paketom a má na starosti funkcie od priradenia paketu k toku až po jeho detekciu nežiadúceho obsahu. Umiestnenie funkcie, ktorá volá manažérsku funkciu popisuje Výpis 4.4.

```
static TmEcode FlowWorker(ThreadVars *tv, Packet *p, void *data)
   {
        FlowWorkerThreadData *fw = data;
        void *detect_thread = SC_ATOMIC_GET(fw->detect_thread);
       ...
        FlowHandlePacket(tv, &fw->fls, p);
       ...
        FlowWorkerStreamTCPUpdate(tv, fw, p, detect_thread, false);
       ...
        AppLayerHandleUdp(tv, fw->stream_thread->ra_ctx->app_tctx,
                          p, p->flow);
       ...
        Detect(tv, p, detect_thread);
       ...
        OutputLoggerLog(tv, p, fw->output_thread);
       ...
        FlowWorkerManager(tv, fw);
   return TM_ECODE_OK;
   }
```

Výpis 4.4: Umiestnenie manažérskej funkcie

Tento prístup nevykonáva ”spánok” vlákna ale funkcia je volaná pre každý paket. Funkcia `FlowWorkerManager` je upravenou verziou pôvodnej manažérskej funkcie, v ktorej prebieha výpočet spánku vlákna na základe pamäťovej vyťaženosti, ktorý je ukladaný do lokálnej premennej `sleep_per_wu`. Podobne prebieha aj výpočet počtu prechádzaných riadkov v jednom volaní funkcie. Aby bolo možné s týmito hodnotami pracovať nielen v jednom volaní, bola pre tieto lokálne hodnoty vytvorená ďalšia štruktúra, ktorá obsahuje všetky pôvodne lokálne premenné ako atribúty. Umiestnenie tejto štruktúry je rovnaké ako vo Výpise 4.3. Analogicky je to riešené pre lokálne premenné funkcie recyklačného vlákna.

Súbor `flow-worker-manager.c` obsahuje redefinície pôvodných funkcii manažéra s upravenou signatúrou a funkcionalitou, ktorá vyhovuje novému prístupu a novým dátovým štruktúram. Z pôvodného riešenia v súbore `flow-manager.c` sa využíva spomínaná funkcia `GetWorkUnitSizing` na výpočet spánku. Táto funkcia bola zmenená o maximálnu dĺžku spánku z 1 sekundy na dlhší čas, pretože tabuliek je viac a predpokladá sa, že sieťová prevádzka bude rovnomerne rozložená medzi tabuľkami. Z tohto dôvodu nie je žiaduce, aby sa vykonávala manažérska funkcia v pôvodných relatívne častých intervaloch. Konkrétna hodnota je predmetom testovania.

V implementovanom riešení sa o čistenie zdieľaných dátových štruktúr starajú všetky `Worker` vlákna. Bolo nutné implementovať zámok, aby nepristupovalo k daným tabuľkám viac vlákien súčasne. Riešenie je popísané vo Výpise 4.5.

```
int FlowWorkerManager(ThreadVars *th_v, void *thread_data)
   {
       ...
        if (SCMutexTrylock(&flow_manager_clean_table_m) == 0) {
           DefragTimeoutHash(fwt->FlowWorkerManagerCounters.ts);
           HostTimeoutHash(fwt->FlowWorkerManagerCounters.ts);
           IPPairTimeoutHash(fwt->FlowWorkerManagerCounters.ts);
           HttpRangeContainersTimeoutHash(
               fwt->FlowWorkerManagerCounters.ts);
           SCMutexUnlock(&flow_manager_clean_table_m);
        }
       ...
   }
```

Výpis 4.5: Pristupovanie k zdieľaným štruktúram

Súčasťou bloku kódu popísaného vyššie je funkcia `DefragTimeoutHash`. Ako bolo spomenuté v Kapitole 3, `Defragmentačná tabuľka` je zdieľanou štruktúrou. V rámci volania funkcie sa čistí len jedna konkrétna tabuľka a to tá, ktorá patrí danému vláknu, ktoré zavolalo funkciu. Nie je to najvhodnejšie riešenie. Môže nastať situácia, že jedna tabuľka sa bude čistiť viac ako ostatné. V najhoršom prípade sa niektorá z nich nebude čistiť vôbec. Toto riešenie vychádza z predpokladu[11], že väčšina sieťovej prevádzky nie je fragmentovaná a s touto tabuľkou nenastáva veľa operácii.

V pôvodnom riešení je pri presúvaní uvoľneného toku do `aside_queue`, volaná funkcia `ProcessAsideQueue`, ktorá spustí činnosť recyklačného vlákna. Tento proces nastáva, ak spomínaná štruktúra nie je prázdna, čiže obsahuje už tok pripravený na kompletné uvoľnenie z pamäti.

V implementovanom riešení miesto, v ktorom sa začne vykonávať recyklačná činnosť zostáva zachované. Opätovne nejde už o plnohodnotné recyklačné vlákno a táto činnosť sa nevykonáva v nekonečnom cykle s prestávkami podľa vyťaženia. Tento proces je súčasťou funkcie `FlowWorkerManager`, ktorá je volaná pre každý paket. Jej činnosť nie je vykonávaná priamo pre každý paket (hlavná časť funkcie je vykonávaná splnenou podmienkou, ktorá je daná časom spánku). Oproti pôvodnému riešeniu, môže v určitých prípadoch (daných charakteristikou sieťovej prevádzky) nastať situácia, že recyklačná činnosť bude vykonávaná častejšie. Tým pádom dochádza k blokovaniu činnosti spracovávania paketov, aj keď pamäť môže byť relatívne voľná. To môže spôsobiť pokles výkonnosti. Aby k tomu nedochádzalo, je recyklačná činnosť obmedzená. Toto obmedzenie vychádza, z aktuálneho využitia pamäte (využitie atribútu `MemcapPressure`[3]). Toto obmedzenie popisuje následujúci Výpis 4.6.

[3]MemcapPressure - udáva percentuálne využitie pamäte pre toky

```
static uint32_t FlowTimeoutHash(FlowWorkerThreadData *fwt,
ThreadVars *tv, FlowTimeoutCountersForWorkerManager *counters) {
{
   ...
    if (td->aside_queue.len >= (100 -
    fwt->FlowWorkerManagerCounters.mp)) {
       cnt += ProcessAsideQueue(fwt, tv, counters);
    }
   ...
}
```

Výpis 4.6: Obmedzenie recyklačnej činnosti

Základné vyťaženie pamäte, ktoré bolo odsledované je od 5 - 7 %. Z podmienky teda vyplýva ak pamäť nie je vyťažená, recyklačný proces je vykonávaný po naplnení približne 93 prvkami. Nastáva problém, že v určitom čase behu systému, v pamäti budú stále uložené tieto nevyužiteľné toky. To či nevyužité toky predstavujú výkonnostný problém je predmetom testovania.

4.1.4 Implementácia Flow tabuľky - RTE_HASH

Súčasťou zadania bolo experimentálne použitie iných dátových štruktúr a nahradenie pôvodných. Implementácia sa teda rozdeľuje na dve optimalizované verzie:

1. Suricata-Optimalizácia (pôvodná hašovacia tabuľka)
2. Suricata-Optimalizácia-RTE_HASH

Obe verzie budú v testovaní porovnávané aký vplyv má na systém použitie iných dátových štruktúr. `Flow tabuľka` je jedna z vyťaženejších dátových štruktúr a preto nahradením rýchlejšej hašovacej tabuľky by mohlo priniesť ďalšie požadované zrýchlenie. Výberu rýchlejšej hašovacej tabuľky predchádzal prieskum na internete. Dôležité parametre tohto výberu boli: rýchlosť a komplexnosť hašovacej funkcie, rýchlosť operácii vyhľadávania a pridávania, možnosť prechádzať iteratívne prvky tabuľky (nie všetky hašovacie tabuľky túto možnosť majú, je to dôležitý parameter z dôvodu čistenia manažérskou funkciou).

RTE_HASH

Vhodným kandidátom, ktorý ponúka vysokú rýchlosť je hašovacia tabuľka **RTE_HASH**[4]. RTE_HASH tabuľka je súčasťou knižnice DPDK. Jej nevýhodou je chýbajúca podpora v režime AF_PACKET. To nepredstavuje problém, pretože testovanie bude prebiehať v DPDK režime. RTE_HASH tabuľka je priamo navrhnutá pre použitie v sieťových aplikáciach. Podporuje napríklad vytváranie vlastného vyhľadávacieho kľúča, niekoľko hašovacích funkcii alebo nastavenie počtu záznamov.

Implementácia je analogická k pôvodnej `Flow tabuľke`. Inicializácia a deinicializácia je volaná v každom `Worker` vlákne. Pri inicializácii je nutné definovať parametre tabuľky: počet záznamov, meno, hašovacia funkcia a ďalšie. Inicializačná funkcia je popísaná vo Výpise 4.7. Kľúč, podľa ktorého sa počíta haš zostáva zachovaný z pôvodnej tabuľky. Kľúč

[4]RTE_HASH -https://doc.dpdk.org/guides-22.11/prog_guide/hash_lib.html

sa skladá zo zdrojovej a cieľovej IP adresy, zdrojového a cieľového čísla portu a použitého protokolu.

```
thread_local struct rte_hash *handle;

static struct flow_key_RTE {
          Address *ip_src;
          Address *ip_dst;
          Port port_src;
          Port port_dst;
          uint8_t proto;
} __attribute__((packed));

void createDPDKFlowHashTable(){
    ...
     params.entries = flow_config.hash_size;
     params.hash_func = rte_hash_crc;
     params.name = table_name;
     params.socket_id = rte_socket_id();
     handle = rte_hash_create(&params);
    ...
}
```

Výpis 4.7: Inicializácia RTE_HASH tabuľky

Práca s novou tabuľkou je definovaná vo funkcii `FlowGetFlowFromHash()`, ktorá bola upravená. Základný koncept fungovania bol zachovaný. Vykonáva sa tu pridávanie do tabuľky v prípade ak paket nepatrí do žiadneho existujúceho toku alebo je paketu priradený existujúci tok.

Ďalej bolo potrebné upraviť čistenie tokov. Do štruktúry `FlowManagerThreadData` bol pridaný atribút `next`, ktorý slúži na uchovanie pozície v rámci prechádzania RTE_HASH tabuľky. Pri čistení je tabuľka prechádzaná spôsobom prvok po prvku za využitia funkcie `rte_hash_iterate()`, ktorá je súčasťou DPDK knižnice. Počet prvkov, ktoré Suricata prejde v jednom volaní funkcie ostáva zachovaný z pôvodnej verzie (hodnota je uložená `FlowWorkerManagerCounters.rows_per_wu`). Rozdielom oproti pôvodnej verzii je, že RTE_HASH tabuľka nemá riadky (resp. `FlowBucket`). Samotné toky sú pridávané a odoberané priamo (bez sekvenčného prechádzania položky tabuľky).

4.1.5 Implementácia Defragmentačnej tabuľky

Defragmentačná tabuľka (`DefragTrackerHashRow *defragtracker_hash`) je štrukturálne veľmi podobná `Flow tabuľke`. Z toho vyplýva, že implementačne ide o podobné riešenie. Je využité kľúčové slovo `thread_local` pre samotnú tabuľku a ďalšie dátové štruktúry, s ktorými tabuľka vykonáva prácu (`DefragTrackerQueue defragtracker_spare_q, DefragContext *defrag_context`). Rovnako boli odobraté zámky z dátových šruktúr.

4.1.6 Implementácia Session pool a Segment pool

Implementácia oboch dátových štruktúr analogicky vychádza z predchádzajúcich riešení upravovaných dátových štruktúr. Ide o použitie kľúčového slova `thread_local` a odobratia zámkov.

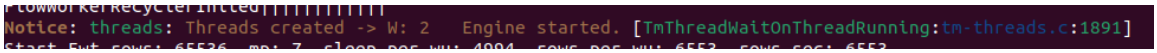
V tomto prípade nejde o tabuľky ale o zoskupenie pamäťové priestoru pre vlákna v poli, indexovaných identifikátorom vlákna. Ideálnym riešením by bolo pristupovať priamo k pamäti, ktorá prislúcha konkrétnemu vláknu. Tento spôsob sa nepodarilo realizovať z dôvodu zložitých závislostí v programe. Pri tomto riešení dochádzalo k nepravidelnému zlému pristupovaniu do pamäte, ktoré sa nepodarilo vyriešiť, pretože dochádzalo k nemu na rôznych miestach programu.

Výsledným riešením teda je, že každé vlákno má vlastný `Session pool a Segment pool`, ktorý obsahuje len jeden prvok a nie sú potrebné zámky. Pri veľkosti samotného systému je predpokladaný rozdiel v implementácii zanedbateľný. Výkonnosť tohto riešenie je predmetom testovania.

4.1.7 Worker vlákno

V súbore `flow-worker.c` sa v implementovanom riešení nachádzajú všetky inicializácie a deinicializácie upravených dátových štruktúr. Nachádza sa tu inicializačná a deinicializačná funkcia manažérskych a recyklačných dátových štruktúr a samotné funkcie pre prácu manažérskej a recyklačnej činnosti.

Zo súboru `suricata.c` a `runmodes.c` boli odobraté funkcie pre vytvorenie, inicializáciu a zobúdzanie manažérskeho a recyklačného vlákna. Implementované riešenie tak neobsahuje žiadne z týchto vlákien. Všetku činnosť vykonávajú `Worker` vlákna.



Obr. 4.1: Činnosť Worker vlákien

Obrázok 4.1 znázorňuje beh systému v režime `Workers`, ktorý obsahuje len samotné `Worker` vlákna.

4.2 Testovanie

4.2.1 Priebežné testovanie

Implementácia prebiehala postupným upravovaním jednotlivých funkčných celkov. To znamená, najprv bola implementovaná napríklad `Flow tabuľka`, potom bola otestovaná jej základná funkčnosť na lokálnom počítači, na ktorom prebiehal aj vývoj. Po overení jej základnej funkčnosti začala implementácia ďalšieho prvku návrhu riešenia.

Testovanie základnej funkčnosti zahŕňalo:

1. **Vizuálna kontrola a preklad súboru** - kontrola pomocou editora a prekladu programu, či sa v kóde nenachádzajú syntaktické chyby a či je možné program preložiť
2. **Porovnávanie výstupu pri spracovaní PCAP súboru** - porovnávanie výstupov PCAP súboru nemodifikovaného kódu s výstupmi už implementovaného riešenia

3. **Kontrolné výpisy na štandardnom výstupe** - pri spracovávaní PCAP súborov boli určité úseky kódu doplnené o výpisy rôznych dátových hodnôt pre lepšiu orientáciu, aký paket bol spracovávaný aj s dátami o jeho uložení do príslušných dátových štruktúr

Pri porovnávaní výstupov boli sledované nasledujúce položky. Prvá sledovaná položka bol celkový počet spracovaných paketov, prípadná stratovosť paketov (packet loss). Ďalšou položkou bol počet uložení do `Flow tabuliek`. Na sledovanie bol použitý PCAP súbor so známym počtom tokov. Vďaka tomu bolo možné odsledovať, či je vytvorený zodpovedajúci počet záznamov v tabuľke a či všetky ostatné pakety patriace konkrétnemu toku boli priraďované do správnej tabuľky. Analogicky prebiehalo testovanie `Defragmentačnej tabuľky`, `Session pool` a `Segement pool`, kde bolo tiež kontrolované správne pristupovanie k dátovým štruktúram.

Pre kvalitné a prehľadné porovnávanie výstupov boli použité rôzne PCAP súbory, v ktorých boli dopredu známe parametre (počet paketov, tokov, TCP spojení, fragmentovaných paketov a pakety so zmeneným poradím). Tieto PCAP súbory boli voľne dohľadateľné na internete. Ak išlo o veľmi špecifickú požiadavku ako má vyzerať daný súbor, bola použitá knižnica PcapPlusPlus[5] a knižnica Scapy[6]. Tieto knižnice umožňujú manipulovať s paketmi rôznymi spôsobmi (fragmentácia a defragmentácia paketov, prevod paketov s adresami IPv4 na adresy IPv6 a ďalšie).

V prvotných fázach bol na tento typ testovania využívaný režim prehrávania paketov podporovaný samotnou Suricatou, v ktorom ako vstupným argumentom bol konkrétny PCAP súbor.

Príklad príkazu pre spustenie Suricaty v režime čítania paketov zo vstupu:

```
sudo ./suricata -c ~/CLionProjects/suricata/suricata.yaml
-s signatures.rules
-r ~/CLionProjects/suricataFiles/BigFlows.pcap
-l /dev/null
```

V neskoršej fáze pri implementácii `Defragmentačnej tabuľky` nebolo možné pokračovať v tomto type testovania. Testovanie nebolo možné vykonávať v režime čítania PCAP súborov. Ako bolo spomenuté v podkapitole 2.4.1, čítanie PCAP súborov zo vstupu pracuje v režime `Autofp`. V tomto režime zachytávanie paketov má na starosť iné vlákno, ktoré nie je `Worker`. Defragmentácia paketov je súčasťou dekódovania paketov, čo má na starosť v režime `Autofp` vlákno, ktoré zachytáva pakety (Recieve thread - prijímacie vlákno). Avšak inštancia defragmentačnej tabuľky je uložená v rámci `Worker` vlákna, do ktorej toto prijímacie vlákno nemá prístup. Z toho vyplýva, že v implementovanom riešení nie je možné používať tento režim, inak dôjde k zlému pristupovaniu do pamäte, teda ku chybe.

Aby bolo možné postupovať podobným spôsobom, bolo nutné využívať už len režim `Workers` a čítať pakety priamo zo sieťovej karty. Avšak sieťová prevádzka priamo zo siete nie je špecifická a je nepredvídateľná. Z tohoto dôvodu je pre účely tohoto typu testovania nevhodná.

Vhodnou náhradou pôvodného testovania bolo vytvorenie virtuálneho sieťového rozhrania na lokálnom počítači. Na vytvorenie virtuálneho rozhrania bol vytvorený krátky skript napísaný v jazyku Bash. Na vytvorené virtuálne rozhranie je možné posielať pakety rôznymi spôsobmi.

[5] PcapPlusPlus - https://pcapplusplus.github.io
[6] Scapy - https://scapy.net

V rámci zachovania pôvodného prístupu testovania, teda čítania paketov z PCAP súborov bol použitý nástroj `Tcpreplay`[7]. Jednou z možností, ktorú umožňuje tento nástroj, je možnosť čítať zo vstupu PCAP súbor a posielať jeho obsah na vytvorené virtuálne rozhranie.

Týmto je možné simulovať v `Workers` režime zachytávanie "reálnej" sieťovej prevádzky, ktorá je upravená podľa našich potrieb (PCAP súbor upravený pomocou nástrojov spomenutých vyššie v tejto podkapitole). Pre vykonanie tohto typu testovania je nutné spustiť program v režime čítania paketov z určitého rozhrania a následne spustiť program `Tcpreplay` v novom terminálovom okne, ktorého argumentom je požadovaný PCAP súbor a rozhranie, na ktoré budú pakety posielané.

Príklad príkazu pre spustenie Suricaty v režime zachytávania paketov zo zvoleného rozhrania:

```
sudo ./suricata -c ~/CLionProjects/suricata/suricata/suricata.yaml
-s signatures.rules -i eno3 -l /dev/null
```

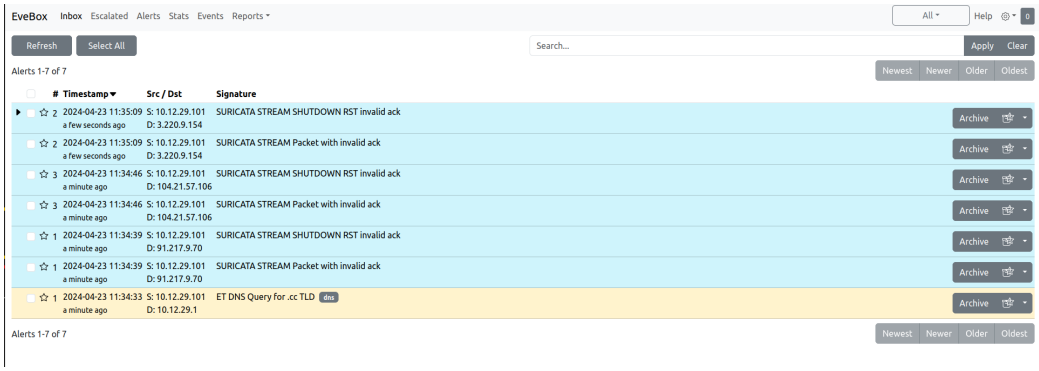
Príklad príkazu pre spustenie programu `Tcpreplay` s požadovaným vstupným PCAP súborom a požadovaným rozhraním:

```
sudo tcpreplay -i eno3 ~/CLionProjects/suricataFiles/pcaps/obama.pcap
```

Súčasťou tohto testovania bola aj kontrola detekčného modulu. Implementované riešenie nezasahovalo priamo do implementácie detekčného modulu. Detekčný modul súvisí s upraveným kódom tak, že využíva dátové štruktúry, ktoré boli upravované na svoju prácu. Súbežne s implementáciou bolo vykonávané testovanie, či nevznikla chyba, ktorá by ovplyvňovala chod detekčného modulu.

Testovanie bolo vykonávané podobným spôsobom. Bolo potrebné použiť PCAP súbor, ktorý obsahoval malvér, alebo iný nežiadúci obsah, ktorý Suricata vyhodnotí ako hrozbu. Takýto PCAP bol získaný z webovej stránky MALVARE-TRAFFIC-ANALYSIS[8]. Následne bola Suricata spúšťaná s argumentom súboru, ktorý obsahuje pravidlá na zachytenie hrozby. Tento súbor je dostupný na stránke Proofpoint Emerging Threats Rules[9]. Po vykonaní týchto krokov a po ukončení Suricaty, bol generovaný výstupný logovací súbor `eve.JSON`, ktorého obsahom sú záznamy obsahujúce upozornenia o nežiadúcom obsahu paketov. Tieto súbory boli porovnávané pred a po implementovaní navrhovaného riešenia, či nedošlo k narušeniu správneho chodu detekčného modulu. Testovanie neodhalilo chyby detekčného modulu viz. Obrázok 4.2.

[7] Tcpreplay - https://tcpreplay.appneta.com
[8] MALVARE-TRAFFIC-ANALYSIS.NET - https://www.malware-traffic-analysis.net/
[9] Proofpoint Emerging Threats Rules - https://rules.emergingthreats.net/



Obr. 4.2: Výstup z testovania funkčnosti detekčného modulu

4.2.2 Testovanie v reálnom prostredí

Testovanie v reálnom prostredí spočíva v nasadení optimalizovanej verzie Suricaty na stroj, na ktorom by bolo použitie Suricaty vhodné. Druhým krokom je meranie dosiahnutého výkonu. V priebežnom testovaní bola testovaná funkčnosť a bol použitý režim AF_PACKET. V tomto testovaní bol použitý režim DPDK, vďaka ktorému je možné spracovávať obrovské množstvo paketov a testovať nielen výkon systému, ale aj jeho funkčnosť, nakoľko je spracovávané veľké množstvo sieťovej prevádzky.

TRex

Na generovanie sieťovej prevádzky v rámci testovania výkonu bol použitý program TRex[13]. TRex je program na generovanie sieťovej prevádzky. Primárne je určený na testovanie sieťových zariadení a protokolov. Je schopný generovať milióny paketov rýchlosťami až niekoľko desiatok gigabitov za sekundu. Podporuje generovanie širokého výberu protokolov (Ethernet, MPLS, TCP, UDP, HTTP, HTTPs, ...). Momentálne je vyvíjaný spoločnosťou Cisco a je bežne používaný na vývoj a testovanie rôznych sieťových aplikácii.

TRex má vlastné python API rozhranie `trex-api`[10] vďaka, ktorému je možné si pomocou skriptu vytvoriť vlastnú špecifickú prevádzku. Podporuje aj prehrávanie PCAP súborov. Pri prehrávaní PCAP súborov je možné si meniť rýchlosti prehrávania, dĺžku a počet opakovaní a ďalšie.

Testovacie prostredie

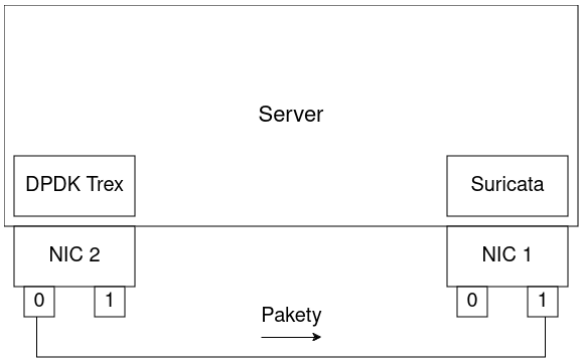
Špecifikácie stroja:

- `OS`: Oracle Linux Server 8.8 x86_64
- `CPU`: Intel Xeon E5-2620 v2 @ 2.1GHz
- `RAM`: 64 GB
- `NIC`:
  1. `2 port Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection` – vstup Suricaty

[10] trex-api - https://trex-tgn.cisco.com/trex/doc/index.html

2. `2 port Intel Corporation Ethernet Controller X710 for 10GbE SFP+` – generovanie sieťovej prevádzky

Na Obrázku 4.3 je znázornená schéma zapojenia. Na sieťovú kartu číslo jedna je pripojená Suricata, z ktorej odoberá pakety. Pakety sú generované programom Trex na sieťovú kartu číslo dva. Sieťové karty sú navzájom prepojené.



Obr. 4.3: Schéma zapojenia

Jednotlivé testy

Testovanie v reálnom prostredí zahŕňalo niekoľko rôznych prípadov testovania:

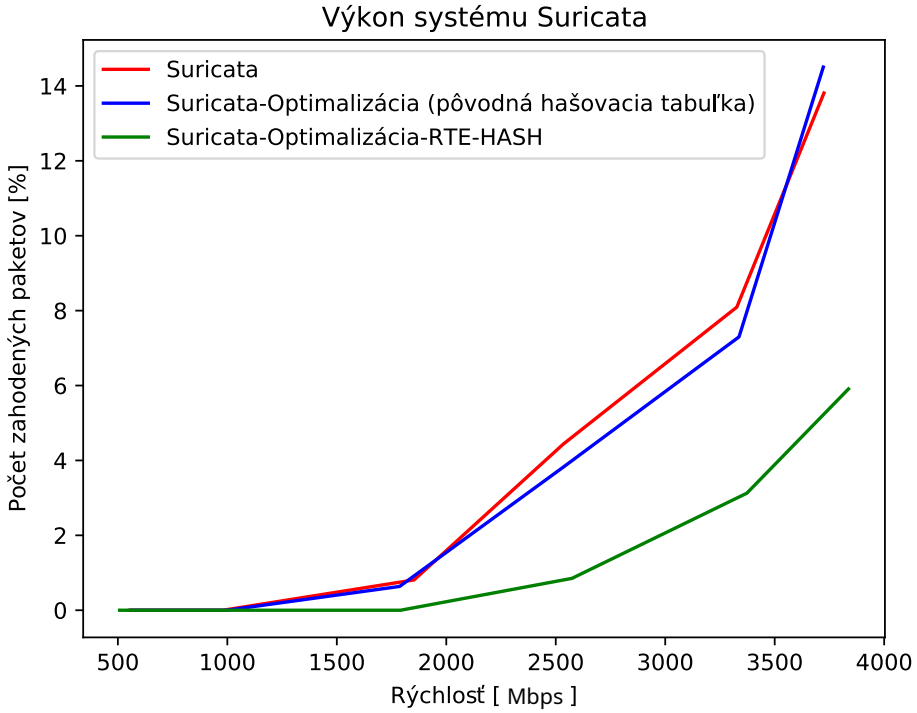
1. Porovnanie celkovej výkonnosti na bežnej sieťovej prevádzke
2. Porovnanie celkovej výkonnosti na rôznych typoch prevádzky
3. Testovanie vhodnej hodnoty času spánku manažérskej funkcie v rámci `Worker` vlákna
4. Testovanie vhodnej hodnoty pre počet čakajúcich tokov na uvoľnenie recyklačnou funkciou v rámci `Worker` vlákna

1. Porovnanie celkovej výkonnosti na bežnej sieťovej prevádzke

Testovanie celkovej výkonnosti systému je najdôležitejší test, ktorý udáva, ako sa systém chová po implementácii navrhovaného riešenia. Výsledky z tohto testu Obr. 4.4 sú hlavným výstupom tejto bakalárskej práce. Poskytujú informácie o dosiahnutí požadovanej optimalizácie a tiež sú podkladom pre ďalšie testy. Reprezentujú vzor chovania optimalizovanej a neoptimalizovanej verzie systému.

Graf na Obr. 4.4 udáva počty zahodených paketov pri rôznych rýchlostiach dátového prenosu. Pri tomto testovaní Suricata používala šesť (`Worker`) vlákien a veľkosti upravených dátových štruktúr boli v pomere 1:6 (to znamená, veľkosť `Flow tabuľky` základnej verzie bola 393216, veľkosť v optimalizovanej verzii 65536, rovnako pre ostatné dátové štruktúry). V danom teste bola Suricata spustená so základnou sadou pravidiel na detekciu hrozieb. Pri teste bez pravidiel na danej sieťovej prevádzke bola dosahovaná stratovosť 0% pri rýchlosti 9.5 Gbps (maximálna prenosová rýchlosť v testovacích podmienkach) na základnej verzii. Z tohto dôvodu nebolo možné testovať variant bez použitia pravidiel.

Sieťová prevádzka, ktorá bola snímaná pochádzala z PCAP súboru, ktorý obsahoval pakety odpovedajúce bežnej sieťovej prevádzke (obsahoval veľa HTTP a HTTPS požiadaviek,



Obr. 4.4: Výkon systému Suricata - charakteristika správania

tie sú pre spracovanie náročnejšie, a sú vhodným kandidátom na testovanie). Tento súbor bol v rámci prehrávania programom Trex, prehrávaný niekoľkonásobne. Tým bol docielený veľký počet operácii pridávania a vyhľadávania v `Flow tabuľke`.

Ak je počet zahodených paketov nad jedno percento, Suricata nedokáže už správne detegovať hrozby, z dôvodu veľkej straty kontextu v rámci tokov. Preto bolo potrebné výsledky normalizovať na relevantné hodnoty (Tabuľka 4.1), t.j. maximálna prenosová rýchlosť pri 0 a 1 % stratovosti.

Testovaná verzia\Zahodené pakety	0%		1%	
	Rýchlosť [Mbps]	Rýchlostný nárast [%]	Rýchlosť [Mbps]	Rýchlostný nárast [%]
Suricata	1550	0	1800	0
Suricata-Optimalizácia	1600	3.2	1900	5.6
Suricata-Optimalizácia-RTE_HASH	2100	35.5	2300	27.8

Tabuľka 4.1: Normalizované výsledky testu výkonnosti

Z Tabuľky 4.1 je možné vidieť mierne odchýlky vo výsledkoch v porovnaní z grafu na Obr. 4.4. Je to spôsobené tým, že pri každom zapnutí Suricaty je výkon odlišný o malé hodnoty. Rovnako graf obsahuje miernu chybu merania, pretože hodnoty boli merané priebežne nie len na konci testovania. Prenosová rýchlosť nie je konštantná, má mierne kolísavý charakter aj 100 Mbps. Môže sa teda stať, že v jednom meraní v určitý okamih môže vyjsť iný výsledok ako v druhom. Konečné hodnoty testovania sú presné. Hodnoty v tabuľkách obsahujú presný výsledok, ktorý pochádza z konca merania.

2. Porovnanie celkovej výkonnosti na rôznych typoch prevádzky

Druhý test bol obdobou prvého testu za použitia iných PCAP súborov. Použité PCAP súbory boli získané z webovej stránky NETRESEC[11]. Testovaná prevádzka bola rôznych protokolov. V tomto prípade obsahovali aj malý počet tokov. Bolo nutné otestovať, ako sa systém chová ak nevzniká veľa nových záznamov v `Flow tabuľke`. Okrem PCAP súborov bola prevádzka generovaná samotným programom TRex s použitím predpripravených skriptov, ktoré sú jeho súčasťou.

Systém bol testovaný aj ako sa správa z dlhšieho časového horizontu (pre tieto účely bol čas testovania 30 minút).

Výsledky testovania ukázali, že systém má približne rovnaké správanie aj pri iných typoch prevádzky. Oproti hodnotám v prvom testovaní (4.1) boli namerané veľmi podobné výsledky. Zo série desiatok testov sa rýchlostný nárast pohyboval v odchýlkach 2 - 3 % oproti hodnotám z prvého testu.

Výsledky z prvého testu je možné považovať za smerodajné a možno ich generalizovať pre rôzne typy prevádzky.

3. Testovanie vhodnej hodnoty času spánku manažérskej funkcie v rámci `Worker` vlákna

V Podsekcii 4.1.3 bolo spomenuté, že manažérska činnosť nie je vykonávaná nad každým paketom. Súčasťou testovania boli rôzne varianty hodnôt. Testovanie vychádzalo z výsledkov prvého testu, to znamená, že testovacím subjektom bola už len najrýchlejšia verzia a tou je Suricata s použitím RTE_HASH tabuľky. Test bol vykonávaný s použitím PCAP súboru z prvého testu. Aby bolo možné zvoliť čo najlepšiu hodnotu, testy boli vykonávané nad rôznymi hodnotami, s rôznym časom prevádzky a rôznou rýchlosťou.

Čas spánku	Čas prevádzky: 60 sekúnd Stratovosť [%]	Čas prevádzky: 10 minút Stratovosť [%]
5 sekúnd	2,8 - 3,4	3,4 - 4,4
15 sekúnd	2,9 - 3,46	3,7 - 4,4
30 sekúnd	2,7 - 3,6	3,8 - 4,5

Tabuľka 4.2: Výsledky testu času spánku manažérskej funkcie pri rýchlosti 2,7 Gbps

Z Tabuľky 4.2 je možné vidieť, že zmena maximálnej hodnoty času spánku nemá vplyv na výsledný výkon (alebo je tento vplyv veľmi malý až zanedbateľný). Ide o o maximálnu dobu spánku. Väčšinu času sa tieto hodnoty podľa rýchlosti prevádzky pohybujú v nižších číslach. Z toho vyplýva, že aj pri vyšších prenosových rýchlostiach, sa bude táto hodnota pohybovať v nižších hodnotách (respektíve v minimálnych). Tento predpoklad dokazuje Tabuľka 4.3.

[11]NETRESEC - https://www.netresec.com/?page=PcapFiles

Čas spánku	Čas prevádzky: 10 minút Stratovosť [%]
5 sekúnd	28 - 31
15 sekúnd	28 - 30,8
30 sekúnd	29,1 - 31

Tabuľka 4.3: Výsledky testu času spánku manažérskej funkcie pri rýchlosti 4,4 Gbps

Z výsledkov testov z Tabuliek 4.2 a 4.3 bolo dokázané, že zmeny hodnoty času spánku manažérskej funkcie majú malý, zanedbateľný až žiadny vplyv na výkon. Na základe toho bola zachovaná pôvodná implementovaná hodnota a to 5 sekúnd.

4. Testovanie vhodnej hodnoty pre počet čakajúcich tokov na uvoľnenie recyklačnou funkciou v rámci `Worker` vlákna

Posledným testom bolo vyhodnotenie, ako často kompletne uvoľňovať už nepoužívané toky (Podsekcia 4.1.3). Tento test bol obdobou tretieho testu. Boli porovnávané rôzne hodnoty počtu tokov pri, ktorom sa spustí recyklačná funkcia. Testovaná bola najoptimálnejšia verzia Suricaty (RTE_HASH). Opätovne bola sledovaná stratovosť pri rýchlejších aj nižších rýchlostiach.

Počet tokov čakajúcich na uvoľnenie	Rýchlosť 2.7Gbps Stratovosť [%]	Rýchlosť 4Gbps Stratovosť [%]
100	2.8 - 3.4	28 - 31
1000	2.94 - 4.21	29 - 30
10000	3.21 - 3.46	30 - 31

Tabuľka 4.4: Výsledky testu počtu tokov čakajúcich na uvoľnenie v recyklačnej funkcii

Výsledky testov dokázali, ako to bolo v teste 3 (4.2.2), že zmena testovaných hodnôt má minimálny až zanedbateľný vplyv na zmenu stratovosti (v konkrétnych testovacích podmienkach). Testovací predpoklad bol, že zmeny v stratovosti by sa mohli prejavovať pri vyšších rýchlostiach, kedy dochádza k prenosu väčšieho objemu dát a zároveň sa musí väčší objem dát čistiť. Predpoklad sa nenaplnil z dôvodu paralelnosti programu. Samotné `Worker` vlákna si prichádzajúcu prevádzku rozkladajú a v jeden moment teda nezvykne byť vykonávaná recyklačná funkcia na všetkých vláknach súčasne.

Na základe výsledkov testovania bola zanechaná pôvodná prvotne implementovaná hodnota 100 tokov. Tento test, tak ako aj test 3 (4.2.2), by bolo možné vykonávať viac do hĺbky a venovať mu väčšiu pozornosť, pre účel tejto práce sú výsledky z testov dostačujúce.

Kapitola 5

Záver

Táto práca bola zameraná na optimalizáciu systému Suricata. Suricata je vysokorýchlostný systém s otvoreným zdrojovým kódom, ktorý plní úlohu IDS alebo IPS systému. V dnešnej dobe narastá dopyt po takýchto bezpečnostných riešeniach a stávajú sa už štandardom. Z tohto dôvodu je čoraz väčší dôraz kladený na výkon, aby tieto systémy boli čo najvýkonnejšie a bolo možné ich nasadiť aj v malých firmách, ktoré nevlastnia najnovšie a najvýkonnejšie vybavenie. Aby bolo možné systémy rozšíriť aj pre tieto skupiny, tak je okrem hardvérového vývoja dôležitá aj softvérová optimalizácia.

Pre dosiahnutie stanoveného cieľa práce bolo nutné dôkladne analyzovať súčasný stav systému a osvojiť si prácu so zdrojovým kódom. Na základe získaných poznatkov bol vytvorený návrh riešenia. Návrh pozostával z rozdelenia zdieľaných dátových štruktúr medzi jednotlivé vlákna programu. Tieto štruktúry boli využívané pri spracovávaní paketov a hľadaní nežiadúceho obsahu. Predpokladom práce bolo rozdelením štruktúr znížiť aktívne čakanie vlákien pri práci s danými dátovými štruktúrami. Z návrhu vychádzali dve implementované riešenia rovnakej optimalizácie za použitia dvoch typov dátových štruktúr.

Prvým riešením bolo vytvorenie vlastnej inštancie, každej z jednotlivých štruktúr z návrhu za použitia pôvodnej `Flow tabuľky`. Od tohto riešenia sa očakával mierny nárast výkonnosti. Druhým riešením bolo vytvorenie vlastnej inštancie jednotlivých štruktúr s použitím vhodnejšej implementácie `Flow tabuľky`, ktorá by priniesla väčší nárast výkonnosti, nakoľko by mala aj optimálnejšie navrhnuté vyhľadávanie v danej tabuľke.

Pri oboch riešeniach bolo nutné ďalej vyriešiť samotné čistenie týchto štruktúr. V pôvodnom riešení existovalo manažérske a recyklačné vlákno, ktoré sa starali o uvoľňovanie položiek z pamäte. V optimalizovaných riešeniach sa o čistenie dátových štruktúr stará `Worker` vlákno, ktoré má na starosti aj spracovávanie paketov a detekciu hrozieb.

Implementáciu nasledoval rad experimentov a testov, ktoré zhodnotili dopad implementovaného riešenia na výkon systému. Testy pozostávali z niekoľkých variant. Najdôležitejším testom bolo skúmanie celkového výkonu na bežnej sieťovej prevádzke. Testovacie prostredie simulovalo reálne prostredie, v akom je bežne Suricata nasadená. Na vstup Suricaty bola posielaná sieťová prevádzka, ktorá obsahovala pakety rôznych protokolov (predovšetkým HTTP a HTTPS).

Meraným výkonnostným parametrom bol rýchlostný nárast pri rovnakej stratovosti paketov oproti pôvodnej verzii. Výsledky tohto testu (Obr. 4.4, Tabuľka 4.1) zistili nárast výkonu pri prvom riešení optimalizácie, t.j. pri zachovaní pôvodných dátových štruktúr 3.2 - 5.6 %. Druhé riešenie, ktoré využívalo RTE_HASH hašovaciu tabuľku dosiahlo výraznejšie zlepšenie. Oproti pôvodnému riešenie bol dosiahnutý výkonnostný nárast v rozmedzí 27.8 - 35.5 %. Dôvodom vyššieho výkonnostného nárastu oproti prvej optimalizácii, bolo

spôsobené architektúrou RTE_HASH tabuľky. RTE_HASH tabuľka je priamo navrhnutá na prácu s tokmi a má rýchle operácie pridávania a vyhľadávania.

Výsledky naznačujú, že implementované optimalizačné riešenia majú potenciál zvýšiť účinnosť systému Suricata. Je potrebné poznamenať, že každá sieťová infraštruktúra je jedinečná a môžu existovať špecifické faktory, ktoré by mohli ovplyvniť úspešnosť implementovaných riešení. Pred nasadením Suricaty do produkčného prostredia je potrebné vykonať ďalšie testy a analýzy. Výsledky tejto práce môžu poslúžiť ako užitočné podklady a inšpirácia pre ďalšie testovanie a budúci výskum v oblasti optimalizácie systému Suricata.

Literatúra

[1] AVIVIANO. *Introduction to Receive Side Scaling* online. Dostupné z: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. [cit. 2023-11-26].

[2] CISCO. *Snort* online. Dostupné z: https://www.snort.org. [cit. 2023-11-11].

[3] GEEKSFORGEEKS. *Socket in Computer Network* online. Dostupné z: https://www.geeksforgeeks.org/socket-in-computer-network. [cit. 2024-04-16].

[4] HYPERSCAN. *About Hyperscan* online. Dostupné z: https://www.hyperscan.io/about. [cit. 2023-11-11].

[5] IBM. *What is an intrusion detection system (IDS)?* online. Dostupné z: https://www.ibm.com/topics/intrusion-detection-system. [cit. 2023-11-25].

[6] IBM. *What is an intrusion prevention system (IPS)??* online. Dostupné z: https://www.ibm.com/topics/intrusion-prevention-system. [cit. 2023-11-25].

[7] OISF. *Home - Suricata* online. Dostupné z: https://suricata.io. [cit. 2023-10-29].

[8] PACIFIC, U. of the. *Raw Packet Sockets* online. Dostupné z: https://ecs-network.serv.pacific.edu/past-courses/2011-spring-ecpe-293b/projects/raw-packet-sockets. [cit. 2024-04-16].

[9] PROJECT, T. Z. *Zeek - An Open Source Network Security Monitoring Tool* online. Dostupné z: https://zeek.org. [cit. 2023-11-11].

[10] PROJECTS, L. F. *Home - DPDK* online. Dostupné z: https://www.dpdk.org. [cit. 2024-04-16].

[11] SHANNON, C.; MOORE, D. a CLAFFY k. *Characteristics of Fragmented IP Traffic on Internet Links* online. Dostupné z: http://conferences.sigcomm.org/imc/2001/imw2001-papers/87.pdf. [cit. 2024-03-27].

[12] TASTYFISH. *Haš tabuľka* online. Dostupné z: https://www.itnetwork.sk/navrh/algoritmy/algoritmy-vyhladavanie/algoritmus-vyhladavanie-has-tabulka. [cit. 2023-11-12].

[13] TEAM, T. *TRex Realistic Traffic Generator* online. Dostupné z: https://trex-tgn.cisco.com. [cit. 2024-04-16].

[14] ŠIŠMIŠ, L. *Optimization of the Suricata IDS/IPS*. Brno, CZ, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Dostupné z: https://www.fit.vut.cz/study/thesis/23479/.