VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

MODUL PRO SLEDOVÁNÍ POLITIKY SÍTĚ V DATECH O TOCÍCH

MODULE FOR NETWORK POLICY MONITORING IN FLOW DATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ADAM PIECEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN WRONA
SUPERVISOR

BRNO 2018

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání diplomové práce

21868

Student: **Piecek Adam, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Modul pro sledování politiky sítě v datech o tocích**
Module for Network Policy Monitoring in Flow Data
Kategorie: Počítačové sítě
Zadání:

1. Seznamte se s možnostmi sledování sítě sondami poskytujícími rozšířená data o IP tocích. Nastudujte nástroje IPFIXcol (sběr informací o síťových tocích) a Nemea (analýza dat o síťových tocích) vyvíjených v organizaci CESNET.
2. Analyzujte obecné potřeby správce na kontrolu politik a nežádoucího provozu v lokální síti.
3. Navrhněte řešení, které umožní správci kontrolovat politiky a odhalovat nežádoucí provoz v lokální síti na základě správcem stanovených pravidel.
4. Navržené řešení implementujte jako modul pro nástroj Nemea.
5. S využitím generovaných i reálných dat o síťových tocích otestujte modul z hlediska využití zdrojů v různých nastaveních.
6. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího zlepšení modulu.

Literatura:

- Dle pokynů vedoucího práce.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce: **Wrona Jan, Ing.**
Konzultant: Žádník Martin, Ing., Ph.D., UPSY FIT VUT
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 26. října 2018

Abstrakt

Cílem této diplomové práce je navrhnout jazyk, jehož prostřednictvím bude možné monitorovat proud síťových toků za účelem detekce porušení síťové politiky v lokální síti. Byla provedena analýza jazyků užívaných v systémech pro řízení proudu dat a dále byl proveden rozbor zadaných úloh od potenciálního správce sítě. Výstupem zmíněné analýzy je návrh jazyka, jenž znázorňuje zřetězené zpracování skládající se z filtrování a agregace. Tyto operace lze přehledně definovat a spravovat v rámci bezpečnostního pravidla. Výsledkem této práce je také modul Policer integrovaný v systému NEMEA, který dokáže aplikovat hlavní příkazy navrženého jazyka. Modul splňuje požadavky zmiňovaných zadaných úloh a může být použit pro další vývoj v oblasti kontrolování síťových politik.

Abstract

The aim of this master's thesis is to design a language through which it would be possible to monitor a stream of network flows in order to detect network policy violations in the local network. An analysis of the languages used in the data stream management systems and an analysis of tasks submitted by the potential administrator were both carried out. The analysis specified resulted in the language design which represents pipelining consisting of filtering and aggregation. These operations can be clearly defined and managed within security rules. The result of this thesis also results in the Policer modul being integrated in the NEMEA system, which is able to apply the main commands of the proposed language. Finally, the module meets the requirements of the specified tasks and may be used for further development in the area of monitoring network policies.

Klíčová slova

Bezpečnost, monitorování síťových toků, analyzování síťového provozu, bezpečnostní politika sítě, síťový správce, návrh jazyka, Systém pro řízení proudu dat, SŘBD, IPFIX, Netflow, NEMEA, Policer, SQL, NFQL

Keywords

Security, Monitoring network flows, Netflow, Network traffic analysis, Network security policy, Network administrator, Language design, Data Stream Management System, DSMS, IPFIX, Netflow, NEMEA, Policer, SQL, NFQL

Citace

PIECEK, Adam. *Modul pro sledování politiky sítě v datech o tocích.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Wrona

Modul pro sledování politiky sítě v datech o tocích

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Wrony. Další informace mi poskytl Ing. Martin Žádník, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Adam Piecek
20. května 2019

Poděkování

Děkuji hlavně svému vedoucímu práce Ing. Janu Wronovi a Ing. Martinu Žádníkovi, Ph.D. za jejich odbornou pomoc a cenné rady, které mi v průběhu práce poskytovali. Děkuji také rodině za podporu po celou dobu mého studia.

Obsah

1 Úvod **3**

2 Monitorování síťového provozu **5**
2.1 Přístupy k monitorování a definice síťového toku 5
2.2 Zachycení síťového provozu sondami 7
2.3 Sběr záznamů IPFIX kolektorem 9
2.4 Analýza síťových toků systémem NEMEA 11

3 Systémy řízení proudů dat v počítačových sítích **13**
3.1 Systémy pro řízení proudu dat 13
3.2 Dotazování nad proudem dat 15
3.3 Jazyk SQL pro proudové zpracování dat 17
3.4 Rozbor nástroje NFQL 20

4 Návrh formátu pravidel pro odhalování nežádoucího provozu **23**
4.1 Komentář k teoretické části 23
4.2 Analýza obecných potřeb správce 24
4.3 Zhodnocení zkoumaných systémů 25
4.4 Navržená struktura jazyka 27

5 Implementace **31**
5.1 Rozhraní modulu 31
5.2 Architektura 32
5.3 Syntaktická a sémantická analýza 33
5.4 Filter 35
5.5 Aggregator 36
5.6 Substituce proměnných 38
5.7 Odlišnosti od návrhu 38

6 Testování a měření výkonu **40**
6.1 Testování 40
6.2 Měření výkonu 42

7 Závěr **46**

Literatura **48**

A Gramatika navrženého jazyka **51**

B Řešení potenciálních potřeb správce sítě **55**

C Obsah CD **58**

Kapitola 1

Úvod

V posledních letech se informační technologie vyvíjejí překvapivě rychlým tempem. Stejně tak roste počet lidí, kteří využívají počítače ke svým účelům. Současné technologie by ovšem nebyly tak populární, kdyby všechny pracovaly izolovaně. Důležitým prvkem je jejich vzájemné propojení, které umožňuje předávání informací. Propojováním jednotlivých zařízení vzniká počítačová síť a následným globálním propojováním počítačových sítí vzniká síť běžně známá pod slovem internet, který přinesl lidstvu nové možnosti a příležitosti. Přes internet komunikují nejenom lidé, ale také ho využívají autonomní zařízení. Ta mohou sloužit pro dobrou věc, ale také mohou být naprogramována tak, aby útočila na ostatní zařízení v síti. Politováníhodným faktem bohužel je, že počet útoků v posledních letech stoupá [20]. Navíc kvůli rozsáhlému propojení přes celou planetu může mít útok velmi široký dopad. Napojení na internet se zdá tedy být jako dvojsečná zbraň. Na jednu stranu poskytuje spoustu pozitivních přínosů, ale na druhou stranu vystavuje zařízení nebezpečí. Většinou však pozitiva daleko převažují negativa, nebo je dokonce připojení tak jako tak nevyhnutelné. Nezbývá než zaměřit úsilí a informační technologie také na bezpečnost počítačových sítí.

Počítačovou síť obvykle spravuje síťový administrátor či správce, který si je dobře vědom bezpečnostních hrozeb, které mohou přijít zvenčí do jeho spravované sítě. Aby měl představu o tom, co se v jeho síti děje, musí nechat síť monitorovat. Na vybraných bodech sítě může správce nechat nainstalovat zařízení zvané sondy, které sbírají informace z proudu dat. Ze sond se poté posbírají informace do jednoho sběrného zařízení zvaného kolektor. Ten poté poskytuje data analytickému systému jako je NEMEA, který dokáže o škodlivém provozu informovat. Díky této infrastruktuře získá správce přehled o své síti a může začít budovat bezpečnostní opatření. Hlavním úkolem této práce je vytvořit v systému NEMEA modul, jenž síťovému správci pomůže identifikovat nežádoucí provoz, čímž se posílí jeho monitorovací prostředky.

Monitorováním sítě se získá nezbytný přehled o tom, jakým způsobem ji uživatelé a připojená zařízení využívají. Tento přehled se poté stává základním kamenem pro definování síťové bezpečnostní politiky, jenž se skládá z kolekcí pravidel určených pro prevenci před bezpečnostními incidenty jako je např. zneužití síťových zařízení či neoprávněný přístup k datům v síti [29]. Pokud počítačovou sítí proudí provoz, který je identifikován bezpečnostním pravidlem jako nežádoucí, tak se o tom eviduje hlášení, jenž se stane podkladem pro protiopatření. Úkolem této diplomové práce mimo jiné je vymyslet, jak by mohlo bezpečnostní pravidlo vypadat v programovacím jazyce.

Návrhu takového jazyka nejprve předcházelo nastudování problematiky monitorování sítě a zpracovávání proudu dat. Tato teorie je rozdělena do dvou kapitol. Kapitola 2 obsa-

huje obecné informace o monitorování a charakterizuje potřebné dílčí komponenty jako je sonda, kolektor síťových toků IPFIXcol či analytický systém NEMEA. Navazující kapitola 3 se zabývá systémy řízení proudů dat a způsobem dotazování. Dále se pokračuje popisem nástrojů určených pro proudové zpracování dat prostřednictvím jazyka SQL. Podrobněji je ovšem rozebrán nástroj NFQL, jehož přehledný programovací jazyk se stal hlavní inspirací této práce. Kapitola 4 se nejprve kriticky zabývá zhodnocením nastudovaných systémů. Dále se pokračuje návrhem jazyka, na který jsou kladeny požadavky plynoucí ze zadaných vzorových úloh. Kapitola končí popisem nově navrženého jazyka, jehož příklady jsou k dispozici v příloze B. Díky jazyku lze definovat bezpečnostní pravidla, jež aplikuje modul Policer nad síťovými toky. Informace o implementaci modulu Policer se nachází v kapitole 5. Následně se kapitola 6 věnuje způsobu testování, výkonnostnímu měření a zhodnocení. Závěrečná kapitola 7 shrnuje diplomovou práci a navrhuje možný budoucí vývoj.

Kapitola 2

Monitorování síťového provozu

V této kapitole se nachází popis významných monitorovacích prvků vypovídajících o tom, jakým způsobem monitorování sítě probíhá. Posloupnost jednotlivých sekcí v této kapitole sleduje, co se děje se síťovým tokem, pokud je předmětem monitorování. Nejprve se musí síťový provoz zachytit nastraženými sondami. Následně se získané informace v podobě záznamů shromažďují v tzv. kolektoru. Ten je schopen dále zpřístupnit záznamy analytickému systému NEMEA. Studie monitorovacích prvků přináší potřebné znalosti o tom, v jakém formátu vstupní data očekávat, z jakého důvodu mají právě takový formát a nakonec v jakém prostředí se bude výsledný modul pro sledování politiky sítě nacházet. Pokud není řečeno jinak, vycházelo se hlavně z prací [14], [31] a [13].

2.1 Přístupy k monitorování a definice síťového toku

Než se přejde k podrobnějšímu popisu jednotlivých fází, je třeba ještě předtím zmínit, že existují dva přístupy, jak se k monitorování postavit. První přístup se nazývá **aktivní monitorování** a je charakteristické tím, že se do sítě zasílají dotazy. Zařízení, jenž je součástí sítě, na dotaz reaguje a z jeho odpovědi se potom vyvozují závěry. Pokud například zařízení neodpoví nebo odpověď nesplňuje očekávání, tak je správce sítě automaticky kontaktován o vzniklém problému. Typicky se tento scénář děje u protokolu SNMP, kdy spuštěný agent na monitorované straně sítě zasílá na výzvy naměřené údaje. Do sítě je také možné zasílat testovací provoz, kterým se zjišťuje například propustnost sítě, ztrátovost paketů či dosažitelnost zařízení. Při aktivním monitorování se tedy zejména využívá znalosti, jak je síť fyzicky uspořádaná a jaké typy síťových zařízení se v jednotlivých uzlech nachází. Nevýhodou tohoto přístupu je, že se síťový provoz takto ovlivňuje a také se může hojným aktivním monitorováním zatěžovat síť.

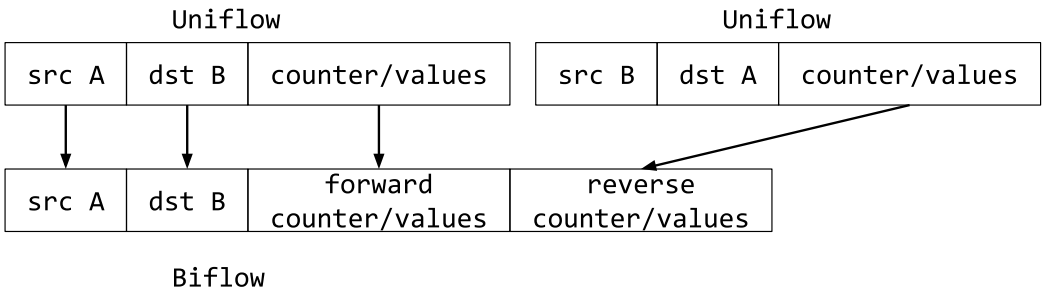
Druhým přístupem je **pasivní monitorování**, které se vyznačuje tím, že se vyhodnocují časové a objemové charakteristiky uživatelského provozu. Monitorovací zařízení analyzuje síťový provoz, který jím prochází, a bez jakéhokoli ovlivnění přeposílá provoz dál. Uživatel sítě tedy vůbec nepozná, že jeho provoz je sledován. Tento přístup se využívá v oblasti bezpečnosti, například pro rozpoznání nežádoucího provozu. Dále nalézá tento přístup uplatnění pro správu sítě nebo při účtování uživatele. Monitorovací zařízení potom zasílá zjištěné informace do sběrného systému, přičemž toto zasílání probíhá autonomně a nikoliv na dotaz sběrného systému, a proto se toto považuje jako součást pasivního monitorování.

Díky přístupu pasivního monitorování je možné zachytávat a analyzovat kompletní pakety ze síťového provozu. Tím jsou typické systémy pro detekci bezpečnostních hrozeb

(Intrusion Detection System, IDS), které v datech paketu vyhledávají známé vzory podobně jako antivirové systémy. Avšak zejména u vysokorychlostních sítí by analyzování kompletních paketů vyžadovalo speciálně výkonný hardware. Navíc je dále třeba mít na vědomí, že síťový provoz může být z velké části šifrovaný, což snižuje účinnost IDS. Proto se používá monitorování síťových toků, které nabízí řešení uvedených problémů. Zachycené pakety ze síťového provozu jsou agregovány do toků (flows) a jsou ukládány pro pozdější analýzu. **Síťový tok** [8] je definován jako:

> Množina paketů procházející bodem pozorování během určitého časového intervalu, kde všechny pakety náležící stejnému toku mají množinu společných vlastností.

Tyto společné vlastnosti mohou být získány ze síťových a transportních hlaviček paketu nebo se také může sledovat obsah z linkové či aplikační vrstvy. Vlastnost může být také odvozená nebo vypočítaná (např. v případě celkového počtu přenesených bytů). Příchozí paket tedy náleží k toku, pokud splní všechny jeho definované vlastnosti, které jsou souhrnně definovány pod pojmem klíč toku (flow key). Ten typicky obsahuje pětici vlastností skládající se ze zdrojové a cílové IP adresy, čísel zdrojového a cílového portu a čísla protokolu. Komunikace mezi dvěma stranami obvykle probíhá ve dvou směrech (A $\rightarrow$ B a B $\rightarrow$ A). Pokud se sleduje komunikace pouze v jednom směru, označuje se potom tento tok jako *jednosměrný tok* (Uniflow, Unidirectional Flow). Obvyklá komunikace typu dotaz-odpověď mezi dvěma stranami je tedy zachycena pomocí dvou jednosměrných toků. Pokud se sleduje komunikace mezi dvěma stranami v obou směrech, označuje se potom tento tok jako *obousměrný tok* (Biflow, Bidirectional Flow) [28]. Komunikace mezi dvěma stranami je potom zachycena pomocí jednoho obousměrného toku, který je kombinací dvou jednosměrných toků, jak je znázorněno na obrázku 2.1.



Obrázek 2.1: Obousměrný tok skládající se ze dvou jednosměrných toků.

Z výše uvedeného plyne, že při monitorování síťových toků se pasivně monitoruje síťový provoz, který se agreguje do podoby jednosměrných či obousměrných toků na základě definovaného klíče toku. Monitorování síťových toků se běžně používá i v sítích s vysokou propustností, jelikož zpracování toků je méně náročnější než zpracování posloupností celých paketů. Navíc se výrazně redukuje objem dat díky agregaci paketů. Pro doplnění je vhodné zmínit další výhody, proč zpracovávat právě síťové toky. Exportování toků je široce rozšířeno v různých síťových zařízení jako jsou směrovače, přepínače a firewally. Poskytovatelé internetu mají v Evropě navíc ze zákona povinnost uchovávat síťové toky po dobu od šesti měsíců do dvou let za účelem vyšetřování, odhalování a stíhání závažných zločinů. Nicméně

z hlediska ochrany soukromí je stále monitorování problematické, jelikož poskytovatelé internetu, výzkumníci a další organizace, mohou v síťových tocích uchovávat také informace z aplikační vrstvy, která může obsahovat citlivé údaje.

2.2 Zachycení síťového provozu sondami

Sonda je specializované zařízení se schopností exportovat naměřená data. V této sekci se rozebírají aspekty se sondou spojené, jako je zachycení síťového provozu, měření, expirace a export záznamu o tocích.

První fáze monitorování síťových toků se zabývá zachycením datového provozu v místě zvaném **pozorovací bod** (Observation Point). Pozorovací bod se nejčastěji nachází v nějakém místě mezi lokální sítí (LAN) a napojením na páteřní síť. Zařízení nasazené na pozorovacím bodě může být přímo napojené na komunikační linku a zachytávat provoz, anebo se může kopie komunikace přeposílat na port, kde je na lince umístěná sonda. Pro zachytávání se typicky používá síťová karta (Network Interface Card, NIC), která automaticky kontroluje, zda při přenosu na komunikační lince nedošlo k poškození paketu. Po zachycení se k paketu přidá *časová značka* (timestamp), protože je součástí základních operací při analytickém zpracování. Časovou značku může přidat jak hardwarová tak softwarová část systému, ovšem v případě softwarové může dojít k časovému skluzu až o několik stovek mikrosekund.

Přidání časové značky je povinná operace provádějící se při zachytávání, avšak obvykle se také provádí **zkracování paketu** či vzorkování a filtrování. Zkracováním paketu se docílí významné redukce množství dat, což má pozitivní dopady pro zkrácení výpočetního času, zmenšení zatížení šířky pásma sběrnice a pro zmenšení paměťových nároků. Pro monitorování síťových toků stačí zachovat hlavičku Ethernetového rámce s hlavičkou síťové a transportní vrstvy.

Množství zpracovaných paketů lze zredukovat kombinací vzorkování a filtrování za účelem výběru zájmové části provozu. Hlavním cílem **vzorkování** je předcházet situaci, kdy dochází k nekontrolovanému zahazování přijímaných paketů, kvůli přetížení zařízení při zpracovávání příliš velkého objemu dat. Díky vzorkování sice dochází ke ztrátě paketů v síťovém toku, ale zahazování probíhá předem definovaným způsobem, tudíž ke ztrátám paketů dochází kontrolovaně. Zároveň je k dispozici stále možnost dopočítat či odhadnout, jak mohl provoz původně vypadat. Největším problémem je, jak vybrat reprezentativní vzorek paketů. K vzorkování je možné přistupovat buď způsobem náhodného výběru, anebo systematicky. Systematické vzorkování rozhoduje deterministickou metodou například periodickým výběrem každého n-tého paketu. Naproti tomu náhodné vzorkování vybírá pakety podle náhodné funkce. Obecně zároveň platí, že čím je vzorkovací frekvence nižší, tím více se redukuje poměr zachycených paketů a tím více se prohlubuje nepřesnost. Přesto se ale ve většině případů volí vzorkování náhodné, protože u něj nedochází k nechtěnému výskytu korelací, jako je tomu u systematického výběru.

Filtrování provozu také redukuje množství paketů, ale na rozdíl od vzorkování, se filtrování snaží oddělit všechny pakety od ostatních na základě jisté vlastnosti. Paket je tedy dále propuštěn do dalších úrovní zpracování na základě splnění deterministických filtračních podmínek zabývajících se obsahem paketu. Zájmové toky nakonec obsahují všechny pakety, které k nim náleží a nedochází tak ke zkreslení jako je tomu u vzorkování. Filtrování lze také provádět dvěma způsoby. Prvním způsobem je paket vybrán na základě stanovené hodnoty v určité části paketu. Ve filtrační podmínce se může také nacházet rozsah hodnot. Typicky se filtruje podle IP adresy, čísla portu atd. Druhým způsobem se aplikuje hašovací funkce

nad určitým obsahem paketu a výsledek se potom porovná s definovanou hodnotou či opět s rozsahem hodnot.

Po zpracování paketů následuje transformace do podoby toků. Jak již bylo uvedeno výše v této kapitole, pro síťový tok se definuje klíč toku sloužící pro agregaci příchozích paketů. Se síťovými toky se pracuje v podobě **záznamů o tocích**, které se dočasně uchovávají v *mezipaměti toků* (flow cache). Záznam se skládá z klíčových a neklíčových částí. Klíčová část definovaná klíčem toku slouží pro identifikaci, ke kterému toku příchozí paket náleží. Dále klíč představuje unikátní identifikátor záznamu v mezipaměti toků, tudíž s příchodem náležících paketů se klíčová část nemění. Naproti tomu neklíčová část se s příchodem náležícího paketu pomocí agregační metody aktualizuje. Vstupem agregační metody je hodnota vyčtená přímo z paketu (např. položka z hlavičky) nebo se může jednat o hodnoty, jenž byly doplněny ve fázi předzpracování paketů (např. časová značka). Výčet sledovaných vlastností může být také doplněn informacemi získatelných z aplikačních protokolů (HTTP, SMTP).

Nový záznam se v mezipaměti toků založí, jestliže se nenajde záznam se stejnou klíčovou částí. Pokud se naopak záznam nalezne, tak dochází k aktualizaci neklíčové části. Záznamy se musí z paměti také průběžně vhodně odebírat a **exportovat**, protože jinak by jistě došlo k zaplnění paměti. Navíc se musí brát v potaz, že toky mohou trvat i v rozmezí několika hodin až dní, takže pokud by nedocházelo k průběžnému odebírání, tak by se o tomto velkému provozu vědělo pozdě. Záznam o toku se exportuje v následujících stanovených případech:

Aktivní časový limit. Pakety odpovídajícího toku přicházejí v očekávaných časových rozestupech, čímž se často aktualizuje záznam o tocích a tok je v tom případě pokládán za aktivní. Pokud doba aktivního toku přesáhne časový limit, pak se hodnoty záznamu exportují, přestože tok ještě stále trvá. Tímto způsobem se oznamuje výskyt dlouho trvajících toků. Pravidelné exportování záznamů dlouhého toku má za následek, že přijímající strana postupně dostává fragmenty dlouhého toku. Záznam se z mezi-paměti toků neodebírá celý, ale pouze se zresetují či vynulují jeho čítače. Typicky se časový limit stanovuje v rozmezí 120 sekund až 30 minut.

Neaktivní časový limit. Pakety odpovídajícího toku se po sledovanou dobu neobjevují a záznam se tedy dlouho neaktualizuje. V tom případě se tok pokládá za neaktivní a není důvod záznam uchovávat v paměti, protože sledovaný provoz může být pravděpodobně stejně tak ukončený. Neaktivní časový limit má význam u transportních protokolů, u kterých nelze jednoznačně identifikovat, kdy je komunikace ukončena (UDP). Časový limit se nastavuje v rozmezí od 15 sekund do 5 minut.

Omezení zdrojů. Speciální heuristiky a adaptivní algoritmy mohou za běhu snižovat hodnoty časových limitů za účelem ušetření svých zdrojů a záznam se tedy exportuje předčasně.

Přirozené ukončení toku. Některé transportní protokoly přenášejí stavovou informaci o ukončení komunikace, takže se záznam podle toho exportuje. Například v případě protokolu TCP se toto signalizuje příznakem FIN a RST. Při opětovném ustanovení komunikace (RST) se neaktualizuje výchozí záznam, ale vytvoří se nový.

Jiné důvody. Může dojít k zaplnění paměti nebo se záznamy manuálně exportují administrátorem.

Obecně lze předpokládat, že pokud jsou časové limity nastaveny na nízkých hodnotách, bude potom kolektor přijímat více záznamů o tocích. Naopak pokud budou časové limity

vysoké, kolektor přijme méně toků, ale na druhou stranu bude dostávat záznamy se zpožděním. Z důvodu snížení zátěže může před předáním záznamu exportnímu procesu docházet k dalšímu filtrování a vzorkování podobně jako tomu bylo u zachytávání paketů.

Exportní proces záznam o toku naformátuje podle zvoleného exportního protokolu. V případě nejstaršího protokolu Netflow v5 je formát jednoznačně určen, tedy jsou definovány jednotlivé elementy a jejich pořadí. Variabilní podobu formátu o toku poskytují protokoly Netflow v9 a IPFIX [21]. V případě IPFIX protokolu nemusí mít všechny exportované záznamy stejnou podobu, protože jak bylo uvedeno dříve, některé záznamy mohou obsahovat informace z aplikační vrstvy. Exportní proces musí nadefinovat strukturu záznamu a přeposlat formát v šablonových zprávách IPFIX kolektoru. Šablona kolektoru sděluje jaké elementy a v jakém pořadí má elementy očekávat. Exportní proces také k šablonám a k nim odpovídajícím datovým záznamům přidává stejné identifikační číslo, podle kterých kolektor pozná, ke které šabloně datový záznam patří.

Jakmile je dokončené formátování záznamu, jsou zprávy následně přenášeny na jeden či více kolektorů pomocí protokolu UPD, TCP či SCTP. Nejrozšířenější pro exportování toků je protokol UDP (User Datagram Protocol), který díky jednoduché režii bývá implementován dokonce i do hardwaru exportního procesu. Na druhou stranu neposkytuje žádnou záruku doručení paketu ani nezaručuje, že při přenosu bude na straně příjemce zachováno pořadí odeslaných paketů. Pro vypořádání se s potencionální ztrátou šablony v protokolu IPFIX je proto doporučené, odeslat šablonu vícekrát v samostatných zprávách. Druhý nejrozšířenější protokol je TCP (Transmission Control Protocol) známý svou spolehlivostí, mechanismy pro zachování pořadí a vyšší režii. Protokol také poskytuje kontrolní algoritmy pro zabránění zahlcení komunikačního kanálu, které nakonec vedou k redukci množství přenášených dat. To má ale neblahý vliv na exportní proces, který je v nejhorším případě nucen uvolnit místo v paměti kontrolovaným zahazováním záznamů. Poslední jmenovaný protokol je SCTP (Stream Control Transmission Protocol) poskytující všechny zmiňované výhody TCP a navíc nabízí alternativu pro řešení zahlcení sítě. Exportující proces může po vypršení časovače přestat čekat na potvrzení přijetí paketu a dá vědět příjemci, ať se ani nesnaží potvrzení posílat. Záznamy mohou být tedy ztraceny, ale na druhou stranu nedojde k tak velkému zatížení exportního procesu jako u TCP a zároveň se šetří zahlcená síť. Další charakteristikou vlastností SCTP je jeho schopnost přeposílat paralelně data více nezávislými kanály. Navzdory výhodám je protokol SCTP méně rozšířený například kvůli jeho náročnější implementaci či chybějící podpoře většiny linuxových distribucí.

2.3 Sběr záznamů IPFIX kolektorem

Následující sekce obecně popisuje problematiku kolektorů a následně se zaměřuje na IPFIX kolektor druhé generace vytvořený v rámci organizace CESNET (sdružení vysokých škol a Akademie věd České republiky)[1]. Kolektor v rámci této práce představuje prostřední článek mezi exportérem toků a analytickým systémem NEMEA.

Úkolem kolektoru je centralizované shromažďování, ukládání, zpracování a následné distribuování záznamů o tocích z exportních procesů. V první fázi **přijímá data** z obvykle více exportních procesů, které mohou být různě pro exportování nakonfigurovány. Kolektor by tedy měl podporovat transportní protokoly UDP, TCP případně SCTP, a také s tím související aplikační protokoly Netflow v5/v9 nebo IPFIX. Stejně tak musí být připraven na kombinace a úskalí vyplývajících z používaných protokolů. Pokud by na kolektoru

[1] www.cesnet.cz

chyběla nutná podpora, došlo by k nevratným ztrátám záznamů, jelikož jsou aplikační protokoly zabývajících se přeposílání toků jednosměrné. Navíc v případě variabilních formátů záznamu musí kolektor řešit správu šablon.

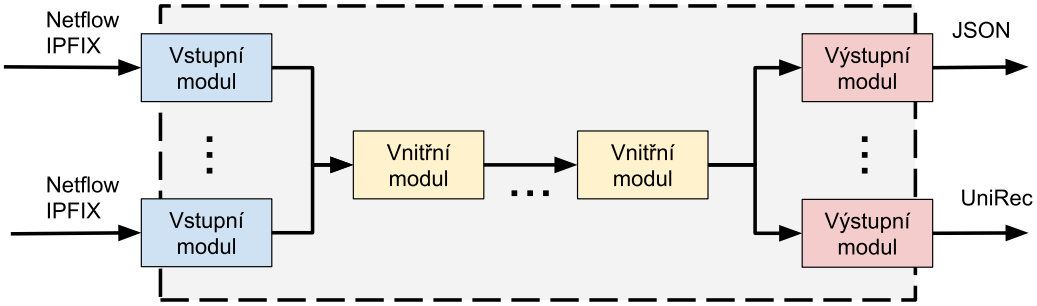
Šablony definují jak přijaté datové záznamy interpretovat. Datový záznam se skládá z tzv. informačních elementů, které se skládají z názvu, unikátního identifikačního čísla, datového typu, popisu a stavu vlastnosti. Informační elementy jsou spravovány autoritou IANA, ale je možné definovat informační elementy vlastní, což může způsobit nekompatibilitu. Dalším důležitým údajem je *identifikátor oblasti monitorování* (Observation Domain ID, ODID) nacházející se v hlavičce IPFIX protokolu. ODID by měl být unikátní napříč všemi zařízeními i napříč jejich spuštěnými exportními procesy. Platnost a rozsah působnosti šablony se váže právě na ODID.

Po přijetí dat následuje volitelná fáze **modifikace záznamů**. Kolektor může k datům přidávat dodatečné informace jako je např. časová značka příchodu záznamu na kolektor či dodá informaci o pravděpodobné geografická oblasti komunikujících stran. Méně obvyklé je generování statistik a přehledových grafů. Kolektor se může uchýlit také k odebírání dat, pokud jsou některá data předurčena za zbytečná či pokud rozpozná v datech chybu. Odebírat se mohou také celé záznamy o tocích, jelikož stejně jako u exportního zařízení může kolektor provádět filtrování či vzorkování. V případě upravování dat lze IP adresy komunikujících stran nahradit jinými IP adresami za účelem zabránění identifikace účastníků komunikace. Toho se dosahuje pomocí algoritmu Crypto-PAn, který bezkolizně mapuje IP adresy a přitom zachovává prefixové vlastnosti adres.

Do poslední a nejdůležitější fáze patří **uložení a distribuce toků**. Typicky se příchozí data přesouvají na pevné disky, pokud nejsou data ihned podrobena analýze, to v tom případě jsou k dispozici v operační paměti. Přístupů pro perzistentní ukládání je celá řada, přičemž se klade důraz na takový formát v databázovém modelu, který poskytuje možnosti rychlého zapisování a zejména čtení pro flexibilní možnosti dotazování. Příklad lze najít u tzv. sloupcových databází, které jsou sice při ukládání komplikovanější, ale to vynahrazují efektivním čtením dat. Sloupcové databáze vykazují také dobré výsledky při kompresi dat. Od kolektoru se očekává, že bude schopen poskytovat data analytickým aplikacím, které např. speciálními detekčními algoritmy odhalí bezpečnostní incident či poskytnou přehledné statistické informace o stavu sítě. Kolektor tedy data poskytuje formou souborů či jiných databázových prostředků, anebo data přímo proudově přeposílá. V prvním případě se ale může stát, že je bezpečnostní incident rozprostřen do více souborů, čímž se může detekce zkomplikovat. Také může nastat časová prodleva než je soubor přístupný pro čtení, po tom co jsou zápisové operace hotové. Při proudovém přeposílání se zmíněné problémy nevyskytují, ale na druhou stranu musí být detekční algoritmy schopné proudově data zpracovávat, což vyžaduje mechanismy klouzavých oken.

Konkrétní implementace IPFIX kolektoru konceptuálně odpovídá popsaným fázím zabývajících se příjmu dat, modifikaci dat, uložení a distribuci toků. Kolektor lze totiž konfigurovat ve třech oblastech, kde v každé oblasti figuruje jiný typ modulu. Vstupní modul spravuje příjem dat, volitelný vnitřní modul modifikuje data a výstupní modul data ukládá nebo je přeposílá pro další zpracování.

Instance modulu se skládá ze jména, identifikátoru a odpovídajících parametrů. V současné době jsou k dispozici vstupní moduly pro přijímaní dat IPFIX protokolem přes transportní protokoly TCP a UDP, vnitřní modul pro anonymizaci IP adres pomocí algoritmu Crypto-PAn, vnější moduly pro rozeslání a uložení záznamů ve formátu JSON, také je podporován formát výstupního souboru kompatibilní s nástrojem nfdump a nakonec je k dispozici rozesílání záznamů v UniRec formátu přes komunikační rozhraní knihovny TRAP.



Obrázek 2.2: Konfigurovatelná architektura zpracování toků systémem IPFIXcol.

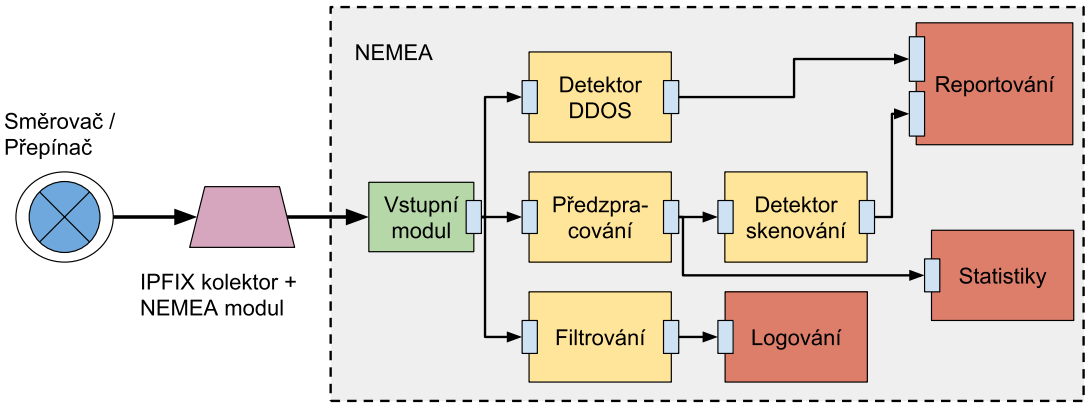
V oblasti konfigurace výstupních modulů lze volitelně doplnit, ze kterých instancích vstupních modulů má výstupní modul čerpat data. To se specifikuje pomocí tzv. ODID filtru. Jak již bylo uvedeno dříve, ODID identifikátor vypovídá o tom, ze které části monitorované sítě a zároveň ze kterého exportního procesu záznamy pocházejí. Díky ODID filtrům lze například shromažďovat toky od různých exportních procesů v různých adresářích.

2.4 Analýza síťových toků systémem NEMEA

Následující sekce vychází z prací [7] a [22]. Proudově orientovaný systém NEMEA (Network Measurements Analysis) je modulární detekční nástroj pro analyzování síťového provozu se zaměřením na síťové toky. Díky proudovému přístupu ke zpracování dokáže systém analyzovat provoz v reálném čase s minimálním počtem čtecích a zápisových operací na disku, což vede k lepšímu výkonu celého systému. Systém NEMEA také poskytuje knihovny tvořící prostředí pro vývoj nových modulů a taktéž prostředí poskytuje mezi nezávislými moduly flexibilní možnosti zapojení. Různé detekční algoritmy implementované v detekčních modulech produkují jednotná hlášení v případě nálezu škodlivého provozu. Systém NEMEA je dostupný ve formě otevřeného zdrojového kódu (open-source) a je součástí kolekce nástrojů STaaS (Security tools as service) zaměřující se na snadnou údržbu a nasazení monitorovacích systémů. Jeden z cílů systému je také zvládnout analyzovat vysokorychlostní síť o rychlosti 100 Gbps. Systém je reálně nasazen v rámci organizace CESNET, kde jsou jeho výstupy využity v systémech Warden a NERD.

Každý modul provádí svůj úkol, jenž může být kupříkladu detekce anomálií, filtrování, agregace, statistické výpočty, vytváření logů či reportů. Příklad **propojení** modulů lze spatřit na obrázku 2.3. Propojení mezi moduly tvoří typicky strom nebo se může jednat všeobecně o acyklický graf s jediným modulem poskytující vstupní data do systému. Zdrojem vstupních dat bývá IPFIX kolektor, ale je možné číst uložené síťové toky z PCAP souboru či odchytávat komunikaci ze síťového rozhraní. Moduly lze psát v programovacích jazycích C, C++ a Python. Systém NEMEA také podporuje dynamické přidávání, odebírání a propojování modulů za běhu programu bez nutnosti přerušovat činnost ostatních modulů.

Veškerá komunikace mezi moduly probíhá přes komunikační rozhraní, které zprostředkovává knihovna **LibTrap**. Ta poskytuje abstrakci pro meziprocesovou komunikaci. Pokud se moduly nachází na stejném serveru, použijí se UNIX sokety. V druhém případě je možné



Obrázek 2.3: Příklad sestavení NEMEA modulů a jejich propojení.

komunikovat přes TCP sokety, tudíž moduly komunikují přes počítačovou síť, což se využívá v případě, kdy je potřeba rozložit zátěž přes více serverů. Komunikační rozhraní modulu se skládá z jednosměrných vstupních a výstupních tzv. IFCs (TRAP Communication Interfaces). Výstupní IFC rozhraní lze propojit s jedním a více vstupních IFCs rozhraní, takže všechny připojené moduly získají stejná data. Komunikační rozhraní IFCs zároveň poskytují kontrolu chyb formátu dat při sestavování komunikačních kanálů mezi moduly. Výstupní IFC rozhraní specifikuje, která datová pole je schopen poskytnout, zatímco vstupní IFC rozhraní specifikuje, která datová pole vyžaduje pro správný chod modulu. Spárování vstupních a výstupních IFCs rozhraní probíhá za běhu systému a formáty jsou automaticky zkontrolovány. Moduly mají navíc možnost pomocí IFC rozhraní zapisovat proud dat do souboru, což bývá užitečné pro pozdější znovu přehrání. Zprávy mohou také obsahovat i data související s hlášením detekce bezpečnostního incidentu.

Moduly zpracovávají příchozí zprávy obsahující informace o síťových tocích v binárním formátu **UniRec**. Tento formát je efektivní pro ukládání a přenos jednoduchých datových struktur, jenž svou podobností připomíná strukturu jazyka C, ale navíc je možné definovat položky proměnné velikosti. Samotný UniRec je generická datová struktura, tudíž konkrétní formát je dán definovanou šablonou, která může být sestavena až za běhu systému. Limitující podmínkou u šablon je, že se vážou k jedinému výstupnímu IFC po celou dobu běhu programu, což ale na druhou stranu vede ke zjednodušení návrhu komunikace. Hlavní výhoda generického přístupu je tedy možnost libovolného rozšíření datové struktury. To má za následek, že i když je systém NEMEA zaměřený na síťové toky, tak lze jednoduše přidat k síťovým tokům informace z aplikační vrstvy.

Skupiny modulů lze centrálně kontrolovat a monitorovat nástrojem s názvem Supervisor, který je implementován formou démona běžícího na pozadí nebo ho lze spustit v interaktivním režimu. Správa modulů se řídí podle konfiguračního XML souboru, v němž se definují názvy spuštěných modulů, jejich parametry a příslušnosti k profilům. Konfigurace může být za běhu programu změněna pomocí konfiguračního protokolu NET-CONF. Pro monitorovací účely získává Supervisor periodicky stavové informace ze spuštěných modulů pro řešení případných problémů. Dále získává statistiky o vytížení výpočetních a paměťových zdrojů.

Kapitola 3

Systémy řízení proudů dat v počítačových sítích

Tato kapitola se zabývá problematikou zpracování proudu dat. Nejprve je vysvětlen charakter proudu dat a architektura systémů pro jejich řízení. Rozebrány jsou také fáze předzpracování, dotazování a posuvná okna. Poté následuje popis dostupných jazyků a systémů pro proudové zpracování. Práce se zaměřila zejména na systémy využívající jazyk SQL a kapitola končí popisem systému NFQL. Zdrojem informací v této kapitole byla prezentace [32], diplomové práce [18], [16] a články [2], [4].

3.1 Systémy pro řízení proudu dat

Proudy dat (data streams) se obecně v informačních technologií objevují tam, kde není možné data uchovávat v tradiční databázi, kvůli jejich obrovskému celkovému objemu. Dokonce to v některých případech ani není žádoucí, jelikož stačí z proudu dat pouze zjistit potřebné informace. To se týká i v případě analyzování síťových toků, které mají charakter proudu dat. Následuje popis vlastností datového proudu dat:

- spojitost — tok dat proudí souvisle.
- uspořádanost — provoz dorazí ve stejném pořadí, ve kterém se generoval.
- proměnlivost — hodnoty se typicky rychle v čase mění.
- rychlost — data se generují rychle a na jejich zpracování je vyhrazený omezený čas.
- velký celkový objem dat — předpokládá se teoreticky nekonečná velikost.

Při zpracovávání proudu dat se musí nasadit **jednoprůchodové algoritmy**, které jsou schopny se vypořádat s dynamicky měnícím se obsahem. Zároveň musí být algoritmy dostatečně rychlé, aby nejlépe vůbec nedocházelo k zahazování dat. Další podmínkou je, že se proud dat zpracovává v reálném čase a zároveň se stíhají aktualizace sumárních charakteristik. Jednoprůchodovost algoritmu je nezbytnou vlastností, jelikož při zpracování proudu dat je vidět každý prvek v proudu nejvýše jednou a navíc se s prvkem pracuje jen omezenou dobu. Pro chod algoritmu se musí také přechodně ukládat úseky z proudu dat, pro které musí být připravena paměť.

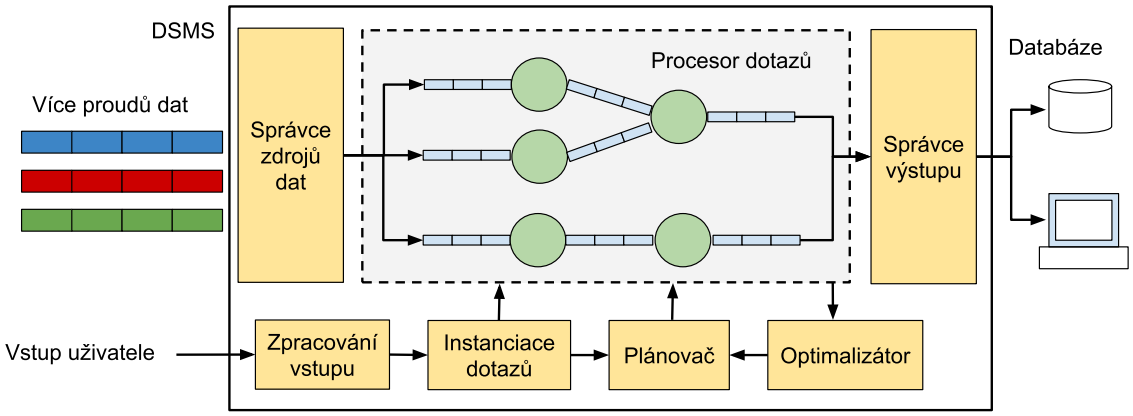
Při vybírání algoritmu je důležité se zabývat tím, jaká je jeho **paměťová náročnost** při zpracovávání proudu dat. V prvé řadě je třeba vyloučit algoritmy nezabývající se horní

hranicí zaplnění paměťového místa. V takovém případě hrozí používání paměťového disku, což vede ke zpomalování celého systému. Algoritmus se musí vybrat takový, aby si vystačil s paměťovým místem v operační paměti. To ale silně závisí na charakteru vstupního proudu dat a typu dotazu. Například pokud není známá potenciální paměťová velikost vstupního proudu dat pro získání přesného výsledku, je nemožné stanovit dostačující hranici paměťového místa pro některé typy operací. Prvním příkladem nechť je operace `JOIN`, která spojuje datové proudy na základě nějakého klíče. Prvek v paměti může ležet neúnosně dlouhou dobu a zabírat potřebné místo. Nezbývá než přidělit operaci nějaký úsek operační paměti a smířit se s nepřesnými výsledky. Operace tak bude pracovat jen s částí proudu dat, i když pro její přesný výsledek potřebuje proud dat celý. Dalším příkladem problematické operace je například výpočet mediánu. Data musí být při klasickém algoritmu k dispozici kompletně v paměti, což ale v případě proudu dat nelze, takže se musí zvolit jiný algoritmus s nepřesným výsledkem. Získá se tak přibližný charakter vstupních dat doplněný kvantitativním vyjádřením, jak moc se lze na výsledek spolehnout. Například se uvádí pravděpodobnost správného výsledku či různé formy míry chyby.

Pro tradiční databáze se používá systém pro řízení báze dat (SŘBD, anglicky DBMS). Kvůli zmiňovaným vlastnostem proudu dat se ale musí s daty pracovat odlišně. To poskytují systémy pro řízení proudu dat (SŘBD, anglicky **DSMS**). Pro zvýraznění rozdílu mezi těmito systémy a uvědomění si omezení při zpracování proudu dat je k dispozici tabulka 3.1. Při analyzování proudů dat se ovšem tradiční databáze využívají a to například pro ukládání získaných sumárních charakteristik.

Tabulka 3.1: Porovnání systému pro řízení báze dat a systému pro řízení proudu dat.

DBMS	**DSMS**
data jsou perzistentní	data jsou dostupná dočasně
přímý přístup k datům	sekvenční přístup k datům
„neomezený“ diskový prostor	omezená operační paměť
důraz je na uložená aktuální data	zaměření na historii dat
bez striktních podmínek na čas zpracování	zpracování v reálném čase
relativně nízký počet aktualizací	vysoká rychlost nových dat
data s různou granularitou	odvozování z nízkoúrovňových dat
přesná data	odpovědi na data jsou nepřesná



Obrázek 3.1: Typická architektura DSMS.

Obrázek 3.1 zobrazuje architekturu DSMS s **procesorem dotazů**. Dotazy bývají v případě proudů dat spojité (Continuous Query, CQ), tzn. že s měnícími se daty na vstupu se zároveň mění výsledek dotazu na výstupu. Výsledkem dotazu může být opět proud dat a v tom případě se odpověď jeví jako spojitá. Na obrázku lze vidět, že DSMS očekává dva typy vstupů. Prvním vstupem se míní proud dat, jenž se může dělit do více nezávislých proudů. Za druhé se od uživatele na vstupu očekávají dotazy vyjádřené dotazovacím jazykem, které se zpracují procesorem a vygeneruje se plán dotazů (query plan). Plán může být vyjádřen ve formě grafu, kde uzly představují operátory (zelená kolečka) např. `SELECT`, `JOIN` a hrany představují pomocné fronty. Každý operátor čerpá data z alespoň jedné své vstupní fronty a svůj výstup ukládá do přiřazené výstupní fronty (sled modrých obdélníků). Z pohledu fronty platí, že jeden operátor ji dodává data na vstupu a z jejího výstupu může číst více operátorů. Dále platí, že každá ze vstupních front operátora musí poskytovat prvek, aby mohl operátor zahájit svou operaci. Aby mohl být plán realizován, musí se nejprve zkontrolovat vstup od uživatele, a poté se v procesoru vytvoří jeho instance. Při běhu DSMS vybírá komponenta *Plánovač* podle definované strategie (např. round robin) operátory či cesty a spouští exekuci. Zpracování dat procesorem probíhá neustále, pokud jsou k dispozici data nebo pokud není jeho činnost zastavena uživatelem. Proudy dat ze vstupu distribuuje komponenta *Správce zdrojů dat* do příslušných cest. Za běhu systému může dále *Optimalizátor* monitorovat procesor nebo zjišťovat, co udělat, aby se například zvýšil výkon a propustnost systému. Výstup DSMS řídí komponenta *Správce výstupu*, který může například ukládat výsledky do databáze či rovnou je zobrazovat uživateli.

3.2 Dotazování nad proudem dat

Tato sekce vychází mimo jiné z prezentace [12] a diplomových prací [25], [15]. Pojmem *dotazy* se v této sekci míní akce zajišťující výběr požadovaných údajů. Jinak řečeno se při dotazování získávají informace, jenž jsou v datech explicitně uvedené a bylo by teoreticky možné si požadované informace dohledat manuálním hledáním. V kontextu analyzování velkého objemu dat se často také používá termín *dolování*. Pojmem *dolování z dat* (data mining) se obecně rozumí získávání znalostí z dat. Jde o automatizované nalezení potenciálně užitečných modelů a vzorů v datech. Úlohou dolování bývá získávání asociačních pravidel a frekventovaných množin, klasifikace a predikce či shluková analýza. Výstupem jsou zajímavé, netriviální či dříve neznámé znalosti, jenž nelze získat nějakým jednoduchým dotazem. Zde vychází najevo rozdíl mezi dolováním a dotazováním. Dolování využívá sofistikovaných metod k získání překvapující znalosti, kdežto při dotazování je snaha najít informace. Jestliže je ale dotaz komplikovaný, získá se nakonec ve výsledku nějaká skrytá znalost a tím pádem je možné označit proces za dolování. V určité rovině se nakonec pojmy dotazování a dolování spojí v jedno a totéž, pokud se tedy dotazuje složitým způsobem.

Při bližším zkoumání také vyplyne najevo, že pojmem dolování z dat se označuje pouze jedna z fází při procesu získávání znalostí. Celý proces se skládá z fází sběru, předzpracování, zmíněného dolování, vyhodnocení a prezentaci výsledků. Fáze jsou relevantní i v případě dotazování, proto následuje jejich výčet:

1. Odstranění šumu a nekonzistentních dat (Čištění)
2. Spojení dat z více zdrojů (Integrace)
3. Výběr relevantních dat pro analýzu (Selekce)

4. Úprava dat do vhodné formy (Transformace)
5. Extrakce vzorů (Dolování, Dotazování)
6. Vyhodnocení vzorů na základě zvolené metriky
7. Prezentace znalostí

Při dolování či dotazování v proudu dat se postupuje po jednotlivých fázích procesu. Protože je proud dat potenciálně nekonečným zdrojem dat, tak všechny fáze procesu mohou pracovat najednou jako při zřetězeném zpracování. Kvůli obrovské celkové velikosti dat totiž není vždy možné v nějaké fázi data ukládat a poté pokračovat v procesu.

Dotazy nad proudem dat se na první pohled příliš neliší od dotazů používaných v DBMS. Přesto existují dvě kategorie dělení, ze kterých vychází najevo, jakým způsobem jsou sémanticky odlišné. V první kategorii se pohlíží na dotazy používané DSMS buď jako na **jednorázové** nebo na **spojité**. Jednorázové dotazy jsou běžně k vidění u DBMS. Dotaz se vyhodnotí v jednom určitém časovém okamžiku nad aktuálním databázovým snímkem typicky na popud od uživatele a poté se zobrazí výsledek. Automatické spouštění dotazu je také možné naplánovat ve více časových okamžicích, kdy obvykle od sebe okamžiky sousedí s určitým časovým rozestupem, případně se dotazy volají periodicky. Spojité dotazy jsou charakteristické pro proudové zpracování dat, protože oproti jednorázovým se spojité dotazy vyhodnocují neustále v souladu s tím, jak přichází nová data. Výsledkem dotazu může být opět proud dat reagující na změny ve vstupních datech nebo je výsledkem jediná odpověď, jenž se na základě vstupních dat průběžně aktualizuje. Jaký formát výstupu se zvolí záleží na konkrétní úloze a preferencí. Například v případě operace `JOIN` se spojují dva proudy dat a vzniká tak kombinovaný proud dat. Při použití operace agregace lze například sledovat aktuální průměr hodnot aniž by uživatele zajímala historie, takže stačí uživateli zobrazovat odpověď měnící se v čase.

V DSMS lze také uvažovat nad druhou kategorii dělení dotazů: **předdefinované** nebo **ad-hoc** dotazy. Pro oba typy dotazů zároveň platí, že mohou být spojité nebo jednorázové. Název předdefinované dotazy napovídá, že se dotazy definují předtím, než se začnou zpracovávat příchozí data. Dotazy se naplánují, přidělí se jim prostředky a pokročilé DSMS mohou dokonce připravit optimalizace. Ad-hoc dotazy se naopak oproti předdefinovaným dotazům zadávají v okamžiku, když už systém nějakou dobu běží a data se zpracovávají. To poskytuje jistou pohodlnost pro uživatele, protože nemusí systém restartovat a tím riskovat ztrátu informací, ale může jednoduše spontánně zadat, co ho aktuálně zajímá. Na druhou stranu však ad-hoc dotazy komplikují návrh a implementaci systému a také je problematická optimalizace těchto dotazů. Dalším problémem je, že uživatel může zadat ad-hoc dotaz a očekávat, že v tíženém budoucím výsledku budou zahrnuta i všechna dosud zpracovaná data. Není ale v tom momentě možné poskytnout přesný výsledek, protože jak bylo zmíněno, prvek se na vstupu objeví jen jednou a neukládá se. Řešením je reprezentovat historická data v nějaké zhuštěné podobě pro poskytnutí alespoň přibližného charakteru. Tato reprezentace dat se označuje jako synopse. Druhým řešením je uživatele informovat o tom, že se ad-hoc dotazy vztahují pouze na data budoucí, tím pádem by bylo možné poskytnout přesný výsledek.

Jak bylo vysvětleno na začátku této kapitoly, nepřesné či přibližné výsledky z dotazů jsou nutnou obětí při používání omezené paměti. Na druhou stranu v mnoha případech přibližné hodnoty výsledků stačí, pokud jsou poskytnuty kvalitně. Obecně se používají konstrukce jako posuvná okna, skeče, náhodné vzorkování, histogramy a další. Z výčtu budou blíže uvedena posuvná okna.

Užívání **posuvného (plovoucího) okna** (sliding windows) je jedna z technik pro získání přibližného výsledku. Dotaz se nevyhodnocuje nad celou historií proudu dat, ale pouze nad jeho poslední částí. Například může stačit zkoumat data za poslední týden pro zodpovězení dotazu, takže data starší než týden se nebudou vůbec brát v potaz. Používání posuvných oken má plnou řadu výhod. Nejprve je vhodné zdůraznit, že se posuvné okno dobře definuje a je jednoduše pochopitelné. Díky jeho jednoznačné sémantice si uživatel může být jistý, jakým způsobem se získala odpověď na dotaz. Dále je výhodou jeho deterministické chování, takže neexistuje žádné nebezpečí, že by se vlivem nešťastného sledu rozhodování docílilo špatné přibližné odpovědi. Nakonec z principu činnosti posuvného okna vyplývá, že se zaměřuje a klade důraz na nedávná data, čímž jde ruku v ruce většině reálných aplikací.

Fundamentálním požadavkem na správnou činnost posuvného okna je, jakým způsobem definovat **časové značky** pro organizaci jejich uspořádání. Časovou značku lze přidat k prvku datového toku, potom se jedná o časovou značku implicitní, nebo se může vybrat atribut prvku, jenž se bude za časovou značku pokládat. Formát časové značky může být klasický unix formát anebo sekvenční číslo. Tyto formáty plní správně svoji funkci, pokud vycházejí z jednoho proudu dat. Problém nastává ve chvíli, kdy se vyskytuje proudů dat více. V této situaci vyvstává otázka, jakou časovou značku má např. operace `JOIN` přidat k výstupnímu proudu dat? Řešením jsou dva přístupy. V prvním případě operace nemusí zaručovat, že její výstup bude podle časových značek seřazen. V druhém přístupu může uživatel definovat, která časová značka má být do výstupního proudu přiřazena.

Výsledek se u posuvného okna generuje po vypršení stanoveného času, které zároveň představuje velikost okna na časové ose. V rámci této práce byly pro demonstraci vybrány zástupci čtyř druhů oken z nástroje Azure Stream Analytics [19]:

- **Tumbling window** rozděluje datový tok na vzájemně se nepřekrývající segmenty. V rámci segmentu se nad daty vyhodnotí funkce a výsledek náleží vždy pouze jednomu segmentu.

- **Hopping window** se posouvá po časové ose v definovaných skocích. Prvky datového toku tak mohou náležet více posunutým oknům. Nastavením stejné velikosti okna a skoku je získáno Tumbling window.

- **Sliding window** produkuje výstup tehdy a jen tehdy, když se objeví prvek v datovém toku. Každý výsledek je docílen minimálně jedním prvkem a okno se posouvá po rozdílu časových značek posledního a předposledního prvku. Prvky mohou opět náležet více posunutým oknům.

- **Session window** je charakteristické svým dynamickým zvětšováním okna. Okno čeká než dorazí první prvek a poté se spustí definovaná časomíra. Jestliže do té doby dorazí další prvek, tak se časomíra znovu spustí a nový prvek je v okně také zahrnut. Pokud se tak nestane, okno se uzavře a objeví se na výstupu výsledek. Okno se tak může zvětšovat po maximální definovanou velikost, kdy se poté násilně ukončí.

3.3 Jazyk SQL pro proudové zpracování dat

Každý systém správy dat by měl uživateli poskytovat flexibilní a intuitivní rozhraní či metody, jejichž prostřednictvím se vykonají dotazy. S nástupem prvních DSMS museli vzniknout nové jazyky poskytující vyjadřovací konstrukce pro operace nad proudem dat. V té

době se vycházelo ze standardizovaného strukturovaného dotazovacího jazyka SQL (Structured Query Language), který patří do rodiny deklarativních jazyků a je přizpůsoben pro práci s relačními databázemi. Deklarativní programování právě poskytuje systému flexibilitu ve výběru optimálních procedur pro vyhodnocení dotazu. Zároveň je uživatel oproštěn od vymýšlení algoritmu, jako je tomu u imperativních jazyků. Zjednodušeně by se dalo říct, že uživatel pouze zadá, jaká data po systému chce a ten mu je nějakým způsobem poskytne.

Jazyk SQL byl podroben **rozšířeními** pro přizpůsobení se požadavkům na zpracovávání proudu dat. Například se modifikovala klauzule `FROM`, aby se dalo odkazovat na proud dat. Dalším rozšířením bylo přidání jazykových konstrukcí pro posuvná okna. Změnila se také drobně sémantika klauzulí `SELECT`, `JOIN`, `UNION` a `MERGE`, u kterých se přidala podpora pro proud dat. Cílem těchto změn bylo, aby se dalo pracovat s proudem dat, jako by to byla tabulka. Vzniká tak nová varianta SQL, jenž se obecně říká *Proudové SQL* (Stream SQL).

U pokročilých rozšíření je možné se také setkat s **kombinování dotazů nad tabulkami dat a proudem dat**. Na proud dat se v tom případě pohlíží také jako na tabulku, akorát je určitým klíčovým slovem poznačeno, že se jedná o proud. Kombinování dotazů je užitečné například k řešení tzv. *pie chart* problému. Pro představu se uvádí tento příklad. Nechť je na webové stránce koláčový graf zobrazující kolik produktů se aktuálně za poslední hodinu prodalo. Graf se má díky proudu informací neustále aktualizovat. Jak by se mohlo přistoupit k řešení problému? Mohla by se ukládat historie všech prodaných produktů a poté by se pravidelným dotazováním vybíraly ty produkty, co se prodaly za poslední hodinu. To ale není efektivní řešení. Prvky z proudu se ale stejně musí někde ukládat, aby poté šlo vyhodnotit dotaz splňující hodinový požadavek. Využít se dá posuvné okno o velikosti jedné hodiny v kombinaci s agregací anebo se může právě explicitně vytvořit relační tabulka s hodinovým souhrnem, jenž se bude průběžně aktualizovat proudem dat.

V rámci této práce byly **hledány nástroje** s otevřeným zdrojovým kódem využívající proudového SQL. Primárně byly vyhledávány nástroje napsané v jazyce C nebo C++, kvůli nativní podpoře těchto jazyků v analytickém nástroji NEMEA. Koncepčně byly však studovány také nástroje napsané v jiných jazycích. Dále tedy následuje popis obecných informací, některých charakteristických rysů a vybraných dosud nezmíněných přístupů nalezených v těchto vybraných nástrojích.

STREAM [3] je projektem Stanfordské univerzity, který po sobě zanechal několik teoretických základů v oblasti proudového zpracování dat a obecnou implementaci DSMS. Dále se navrhl deklarativní dotazovací jazyk CQL, jelikož rozšířené SQL se v komplexnějších dotazech jevilo nepřehledné, přesto je ale syntakticky podobné. Definované byly operátory pracující s relacemi obohacené časovou značkou i proudem dat. Operátor může obsahovat tzv. *synopse*, ve které se ukládá stav, jenž může být vyžadován pro budoucí vyhodnocení operátoru. Nezbytně se dále synopse využívají v posuvném oknu, poddotazování či pro mezi-výpočet nepřesného výsledku dotazu. Jak bylo zmíněno v této kapitole, na základě dotazů se vytvoří plán v podobě grafu a synopse se tedy nachází v jeho uzlech. Při zkoumání synopsí se došlo k poznatku, že v jednom dotazu jich více obsahuje totožná data. Redundance je vyřešena nahrazením synopsí za konstrukcemi *stub* a *storage*. Stub vlastní stejné komunikační rozhraní jako synopse a lze chápat jako ukazatel na místo v paměti neboli storage. Sdílení paměti je možné za běhu systému povolovat a invalidovat. Projekt STREAM se nedodělal úplně do konce a v roce 2006 byl oficiálně ukončen jeho vývoj, takže v jeho poslední verzi zůstalo několik omezení. Nelze v klauzuli `WHERE` definovat poddotazy, chybí podpora klauzule `HAVING`, v klauzuli `SELECT` nelze definovat aritmetické výrazy, atributy mohou nabý-

vat pouze čtyř datových typů, další omezení patří časovým oknům atd. Neoficiálním novodobým vzkříšením tohoto projektu by mohl být systém Calcite SQL[1], ve kterém je znát, že se projektem inspiroval a přidal další vylepšení, ale i ten se dosud potýká s nedokončenými cíli.

TelegraphCQ byl vyvinut v rámci UC Berkley's Computer Science Division a je součástí projektu Telegraph. Pro svou činnost bylo DSMS integrováno do DBMS PostgreSQL, jehož některé funkce byly přizpůsobeny pro proudové zpracování dat. Cílem bylo také vytvořit infrastrukturu typu klient server, kde klient zadává dotazy a další příkazy serveru, který má přístup k proudu dat a spravuje DBMS. Databáze je k dispozici, aby bylo možné archivovat zájmové proudy dat či slučovat informace z databáze a proudu. Cílem také bylo optimalizovat dotazy od všech klientů a sdílet prostředky dotazů. Stejně jako v případě projektu STREAM byl vývoj ukončen kolem roku 2006. V návrhu byly učiněny chyby a kupříkladu chybí podpora některých jazykových vyjádření. Nejkritičtější je nejspíš absence podpory operátoru `OR` a `NOT` v klauzuli `WHERE`. V rámci diplomové práce [26] se zkoumala detekce zaplavení oběti SYN útokem a kvůli jazykovým limitům bylo vymyšlení dotazu náročné.

Stream Mill [27] je obecné DSMS využívající jazyk ESL (Expressive Stream Language) podobný SQL. Zajímavou možností tohoto systému je definovaní uživatelských agregátů pomocí konstrukcí UDA (User Defined Aggregates). UDA se skládá ze tří programových bloků `INITIALIZE`, `ITERATE` a `TERMINATE`. V prvním bloku se definují operace, jenž se provedou při příjmu prvního prvku z proudu dat či před započtením zpracování dat. V druhém bloku `ITERATE` se provádí činnosti nad zbývajících prvcích a třetí blok `TERMINATE` se provede při zpracování posledního prvku nebo voláním. Uvnitř bloku se manipuluje s proměnnou či stavem agregátu a také s interními tabulkami do nichž se ukládají příchozí data. Bohužel zdrojové kódy tohoto systému nebyly v době psaní této práce nalezeny.

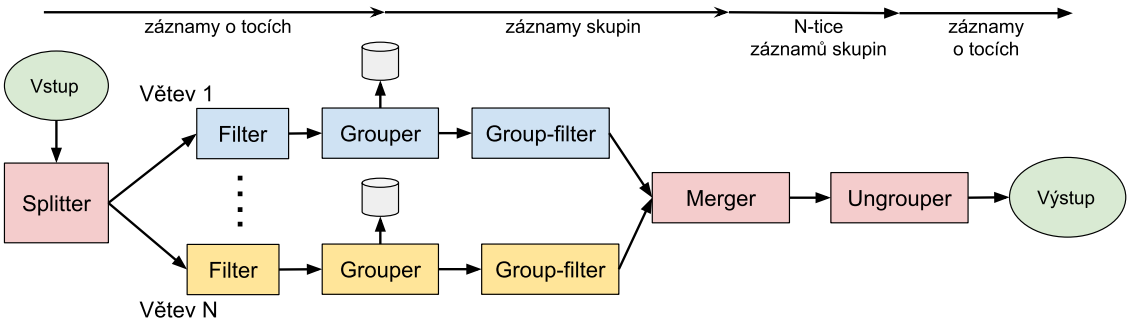
WSO2 [9] [24] lze zařadit do nové generace softwarových proudových procesorů a nástrojů s otevřeným zdrojovým kódem. Od roku 2016 se objevují nástroje jako Apache Storm, Flink, Kafka, Samza či Beam, aby byly dostupné technologie pro vypořádání se s problémy okolo zpracování Big data. Platforma WSO2 je celosvětově rozšířena a používána mnoha organizacemi. Přijímat data lze z Kafka Stream, HTTP protokolu, zpráv od brokera apod. Pro dotazování je k dispozici jazyk SQL s podnázvem Siddhi, ve kterém lze provádět operace jako filtrování, agregace a sumarizace, manipulace s okny, korelace událostí aj. Uživatel může pracovat s připraveným IDE nástrojem Stream Processing Studio, v němž může psát SQL dotazy nebo doplňovat aplikace napsané v jazycích Java nebo Python. Proudový procesor sbírá události, detekuje v nich pomocí SQL dotazů vzory a dokonce je možné spustit predikce pomocí strojového učení. O výsledcích generuje hlášení, které lze živě vizualizovat pomocí dashboards. K dispozici je dále ke stáhnutí více než 50 rozšíření pro specifické případy užití jako je zpracování geografických dat, přirozeného jazyka, strojového učení, detekce anomálií a dalších, ale zatím není podpora pro síťové toky. WSO2 dále poskytuje možnost ukládání dat do různých úložných systémů jako RDBMS databáze, Apache Solr, MongoDB, Apache Hbase aj.

[1] https://calcite.apache.org/

Stream4Flow [17] je framework pro analyzování IP toků v reálném čase využívající distribuovaného zpracování pomocí systému Apache Spark Streaming. Stream4Flow funguje tak, že z IPFIX kolektoru proudí data ve formátu JSON do systému Kafka umožnující rozesílání zpráv. Role frameworku Apache Spark je v proudovém zpracování příchozích zpráv pomocí modulů popsaných v jazyce Scala, Java či Python a pomocí programového modelu MapReduce. Stejně jako v předchozím případě systému WSO2 jsou k dispozici různé možnosti a technologie pro ukládání dat a jejich vizualizaci. Uživatelské moduly využívají RDD API, jehož programovací konstrukce na první pohled nepřipomínají jazyk SQL, avšak pomocí knihovny Spark SQL lze poté doplňovat SQL dotazy. Ve zdrojových kódech frameworku Stream4Flow se nachází připravené detekční moduly[2] pro detekování útoků jako DDOS, skenování portů či útok na SSH autentifikaci.

3.4 Rozbor nástroje NFQL

NFQL [6] [5] je nástroj a zároveň dotazovací jazyk, jenž je přímo zaměřen na proudové zpracování síťových toků. Motivace vytvoření tohoto nástroje spočívala v roce 2009 na stanovisku, že nástroje využívající jazyky založené na bázi SQL mohou vést k nízké výkonnosti. Podobně filtrovací jazyky používané v nástrojích jako nfdump[3] či flow-tools[4] zatěžují systém a navíc postrádají funkci souběžného zpracování dat. Procedurální jazyk nacházející se např. v nástroji Silk[5] je velmi užitečný, ale není zrovna triviální na pochopení. Vymýšlel se tedy nový přístup se zaměřením na identifikaci vzorů v síťových tocích s možností definovat ve vzoru časové závislosti. Vznikl tak deklarativní dotazovací jazyk s důrazem na jeho jednoduchost a časové závislosti jsou řešeny s pomocí tzv. Allen's time interval algebra[6].



Obrázek 3.2: Schéma zřetězeného zpracování NFQL.

Oproti ostatním jazykům se jeví NFQL na první pohled přehledné, díky jeho popisu zpracování dat pomocí několika paralelních větví a stádií. Větev představuje zřetězené zpracování dat rozdělené až do šesti stádií. Na obrázku 3.2 je vidět schéma zřetězeného zpracování. Dřív než dojde k popisu jednotlivých stádií je třeba předtím popsat druhy porovnávání:

[2] https://github.com/CSIRT-MU/Stream4Flow/tree/master/applications/detection
[3] https://github.com/phaag/nfdump
[4] https://linux.die.net/man/1/flow-tools
[5] https://tools.netsa.cert.org/silk/
[6] https://en.wikipedia.org/wiki/Allen%27s_interval_algebra

Absolutním porovnáváním je míněno, že se na pravé nebo levé straně relačního operátoru nachází identifikátor atributu a na opačné straně se potom musí nacházet konstantní hodnota odpovídajícího typu např. IP adresa nebo číslo portu. Atribut je získán ze současně zpracovávaného záznamu o toku.

Relativní porovnávání označuje, že se na pravé i levé straně relačního operátora nachází identifikátor atributu. Porovnávají se tedy hodnoty atributů ze dvou záznamů.

Tělo stádia se skládá z jednoho či více termů, jejichž kolekce tvoří klauzuli. Term může obsahovat jedno či více absolutních nebo relativních porovnání, ale musí být mezi nimi binární operátor *OR* (disjunkce). Mezi termy se implicitně nachází binární operátor *AND* (konjunkce). Klauzule obsahující termy je uzavřená ve složených závorkách. Dále platí, že mezi klauzulemi se implicitně nachází disjunkce. Celá větev tudíž obsahuje stádia, jenž dohromady tvoří disjunktivní normální formu (DNF). Nyní následuje obecný popis jednotlivých stádií.

Splitter zprostředkovává vstupní data systému jednotlivým větvím. Pro distribuci platí, že každá větev dostane identickou kopii záznamu o toku, aby mohlo zpracovávání ve větvích probíhat nezávisle. Tělo stádia Splitter zůstává prázdné, jelikož žádné klauzule pro svou činnost nepotřebuje. Pokud stádium chybí, tak se implicitně doplní.

Filter obsahuje v těle klauzule, kde se provádí absolutní porovnávání. Pro záznam platí, že pokud splní všechny podmínky filtrování, pokračuje do dalšího stádia.

Grouper slučuje záznamy o tocích do skupin podle nějakého klíče. Klíč je vyjádřen pomocí klauzulí obsahující absolutní nebo relativní porovnávání. Při seskupování se může provádět agregace a její výsledek se přidá jako atribut k datové struktuře skupiny. Záznam o toku musí také splnit časové podmínky, aby mohl být přidán do příslušné skupiny. Například pro seskupení záznamů lišících se maximálně o 500 milisekund se napíše term doplněný o *delta 500 ms*. Dále platí, že pokud není možné záznam přidat do nějaké skupiny, potom se zakládá nová skupina obsahující tento záznam.

Group-filter obsahuje absolutní porovnávání pro filtrování skupin ze stádia Grouper. Opět platí, že skupina musí splnit všechny podmínky, jinak je zahozena.

Merger spojuje skupiny z protějších větví na základě relativního porovnávání a také na základě splnění časového intervalu definovaného pomocí Allen's time interval algebra. V podstatě se opět jedná o seskupování jako tomu bylo ve stádiu Grouper, ale s tím rozdílem, že se seskupují skupiny, neprovádí se agregační funkce a navíc lze definovat podmínky časových intervalů. Výstupem je proud skupin poskládaných do spolu souvisejících celků. Zjednodušeně řečeno se ve stádiu Merger nějakým způsobem uspořádají skupiny pocházející z více větví do jedné posloupnosti.

Ungrouper rozbaluje skupinu zpátky do jednotlivých záznamů o tocích a ukládá je na disk. Při použití stádia Ungrouper se tedy nenacházejí ve výstupním souboru záznamy s agregovanými hodnotami, ale záznamy o tocích, jenž se vyskytovaly na vstupním rozhraní systému NFQL. Stádium stejně jako Splitter nepotřebuje žádné klauzule, ale na rozdíl od něj se implicitně nedoplňuje. Pokud tedy stádium chybí, tak rozbalování skupin neproběhne.

```
branch A {
    filter F { dstPort = 80
               protocol = TCP }
    grouper G { srcIPV4 = srcIPV4
                flowStart = flowStart delta 500 ms
                aggregation { sum(PacketDeltaCount }}
    group-filter GF { packetDeltaCount > 200 }
}
branch B {
    ...
}
merger M { A.srcIPV4 = B.srcIPV4
           A.PacketDeltaCount < B.PacketDeltaCount
           A o B OR A f B }
ungrouper U {}
```

Nechť je popsáno shrnutí činnosti systému. Uživatel sepíše zřetězené zpracování v několika větvích a doplní potřebná stádia. Nechť uživatel použije všechna dostupná stádia. Filtrováním se vyberou pouze zájmové toky, které se následně agregují podle definovaných podmínek a přidají se hodnoty agregačních funkcí. Zpracování pokračuje dále zkontrolováním agregačních hodnot ve stádiu Group-filter. Výstupy z větví lze propojit do jednoho toku pomocí stádia Merge a nakonec lze získat původní síťové toky bez stopy agregace. Výstupem systému jsou tedy síťové toky, jenž se dostaly skrz dvojí filtrování, a které jsou navíc seskládány k sobě podle klíče ze stádia Grouper či Merger.

Nástroj NFQL se implementoval dohromady dvakrát. První verze nesla název Flowy a byla implementována v jazycích C a Python roku 2010. Jelikož rychlost zpracovávání dat nebyla dostatečná, postupně se kód přepisoval z jazyka Python do jazyka C. Druhá verze byla implementována v roce 2013 už s označením NFQL. Jádro systému je kompletně přepsáno do jazyka C a jazyk Python zůstal pouze pro parsování jazyka. Systém zvládne zpracovávat vstupní data ve formátu Netflow v5 a IPFIX. Od roku 2013 systém neprodělal žádnými aktualizacemi a aktuálně obsahuje kolem 50 problémů. Parser jazyka prakticky není se systémem integrován a nevypadá to, že by se na vývoji pokračovalo.

Ač autoři označují NFQL slovy stream-based flow query language, tak při detailnějším zkoumání implementace systému bylo zjištěno, že program nezpracovává data proudovým způsobem, jak bylo popsáno na začátku této kapitoly 3. Především systém očekává na vstupu soubor s uloženými daty nikoli rozhraní pro příjem proudu dat. Systém postupně přečte celý soubor a uloží si jen ty záznamy, které splňují podmínky filtrace. Jakmile je s tímto hotov, pokračuje zpracováváním dat.

Kapitola 4

Návrh formátu pravidel pro odhalování nežádoucího provozu

V předchozích kapitolách byly popsány prostředky pro monitorování sítě a dále byly charakterizovány dosavadní systémy pro zpracování proudu dat. Tato kapitola se v úvodu zabývá potencionálními potřebami správce sítě, aby se specifikovaly minimální požadavky na výsledný modul pro systém NEMEA. Dále se rozebírají systémy pro řízení proudu dat, jenž byly probrané v předchozí kapitole. Poté na základě předchozích rozborů vyplývá potřebný formát pravidel, který je v závěrečné podobě vypracován do jazyka a jeho popis se nachází na konci této kapitoly.

4.1 Komentář k teoretické části

Výsledkem této práce má být zlepšení systému NEMEA v oblasti odhalování nežádoucího provozu pomocí funkčního modulu, který zároveň uživateli poskytne nové vyjadřovací možnosti v rámci definování detekcí. Systém NEMEA musí zprostředkovávat modulu vstupní data pomocí komunikačního rozhraní knihovny LibTrap. Systém NEMEA může vstupní data čerpat z uloženého souboru obsahující síťové toky, anebo lze data odebírat přímo z IPFIX kolektoru.

Z popisu monitorovacích komponent v kapitole 2 plyne, že se nelze spolehnout na úplnost poskytovaných dat, jelikož síťový provoz může představovat obrovský objem neustále protékajících dat, což vytváří striktní požadavky na zpracování. Nejkritičtějším článkem může být monitorovací sonda určená pro zachytávání a exportování síťových toků. V prvé řadě může sonda během zachytávání síťového provozu využívat mechanismy vzorkování a filtrování. Nejhorší dopad na kvalitu dat činí vzorkování, protože během tohoto procesu může dojít k zahození důležité části pro detekci bezpečnostního incidentu. U filtrování je situace lepší, pokud je tedy k dispozici informace, jakým způsobem se síťový provoz filtroval. Další komplikací pro detekování může být způsob, jak dochází k exportování záznamu o toku, protože definované časové limity mohou rozdělit jeden síťový tok do více záznamů. Dokonce ani v případě přenosu záznamu o toku z monitorovací sondy ke kolektoru není zaručena bezztrátovost informace, jelikož se pro přenos nejčastěji používá transportní protokol UDP.

Na monitorování sítě je také zajímavé se podívat z pohledu dotazování nad proudem dat. V sekci 3.2 byly vypsány hlavní fáze, jenž se provádí při procesu získávání znalostí. Odstranění šumu a nekonzistentních dat provádí z velké části monitorovací sonda a částečně se na tom může podílet také kolektor. Fáze jako integrace z více zdrojů dat, selekce

relevantních dat pro analýzu a transformace do vhodné formy, je už pouze zodpovědnost kolektoru. Další fáze jsou ponechány na systému NEMEA. Cílem této diplomové práce je vytvoření modulu, který se zabývá dotazováním v podobě detekcí a vyhodnocením zvolených metrik, jenž rozhodnou o tom, zda učinit hlášení anebo nikoliv. Částečně by se také měl modul podílet na fázi prezentace, jelikož hlášení může být předáno definovanou formou dalšímu modulu či systému, jenž se fázi prezentace plně věnuje.

4.2 Analýza obecných potřeb správce

Obecné potřeby správce byly zadány po konzultaci s Martinem Žádníkem a s vedoucím této práce Janem Wronou. Následuje výčet znění jednotlivých potřeb, které jsou navíc doplněné komentářem a informacemi. Ke všem detekcím se zároveň implicitně předpokládá, že je k dispozici možnost definování časového intervalu např. v podobě posuvného okna.

1. **Detekce spamu**: *z dané podsítě nesmí žádná IP adresa přistupovat k více než X SMTP serverům.*
   Už z prvního zadání plyne, že v detekčním pravidlu bude muset existovat možnost, jak definovat libovolnou podsíť ze spravované sítě. Detekce souvisí s nežádoucím množstvím rozeslání elektronické pošty (e-mailů). SMTP server může klasicky naslouchat na portu 25, kde očekává doručení zprávy od jiného serveru. Dále naslouchá na portu 587, ke kterému se obvykle přistupuje pomocí šifrovaného protokolu TLS. Tento port využívají poštovní klienti. Komunikace dále obvykle probíhá s použitím transportního protokolu TCP. Detekční pravidlo by mělo také sledovat překročení stanovené meze *X*. Volitelně by měla být přítomna možnost počítání unikátních připojení k SMTP serverům.

2. **Detekce reflektorů**: *žádná IP adresa nesmí odesílat příliš mnoho velkých paketů.*
   V technologii Ethernet se IP datagram přenáší s maximální velikostí označovanou jako MTU (Maximum Transmission Unit), jehož obvyklá hodnota je 1500 bytů. Detekce velkých paketů by se mohla ze síťového toku určit např. pomocí průměrného počtu přenesených bytů v jednom paketu a poté by následovalo porovnání s definovanou mezí. Na správci sítě by tedy bylo definování příslušné meze, kterou by mohl zjistit pomocí nějakého analytického nástroje. Alternativou by mohlo být dynamické zjišťování aktuálního stavu sítě, kde by se velikost příliš velkých paketů odvodila. To by pro správce znamenalo, že by nemusel zadávat konstantní hodnotu pro mez.

3. **SIP flood**: *na ústřednu nesmí jít více než X SIP příkazů s příznaky REGISTER nebo INVITE.*
   V tomto případě musí být v síťových tocích k dispozici informace z aplikační vrstvy. Konkrétně musí být přítomna informace o příznacích signalizačního protokolu SIP, jenž se používá pro sestavení spojení před přenášením hlasu v digitální formě. Útočník může zahltit server vysokým počtem příkazů REGISTER či INVITE poslaných za krátkou dobu. Požadavek REGISTER zajišťuje registraci účastníka a požadavek INVITE se používá pro založení dialogu.

4. **Peer2Peer**: *žádná IP adresa nesmí komunikovat s více než X adresami na dynamických číslech portů*
   V komunikaci typu klient-klient spolu komunikují přímo jednotliví uživatelé. Tento typ komunikace se používá v aplikacích jako je sdílení souborů či telefonování. Velký

počet různých spojení z jedné IP adresy může například indikovat šíření počítačového červa v podsíti. Z požadavku vyplývá, že by se pro každou IP adresu měla počítat suma různých spojení a opět se má hlídat překročení meze.

5. *Detekce komunikace s adresami na* **blacklistech**.
   V tomto případě se překročení meze hlídat nemusí, jelikož obvykle se netoleruje žádná komunikace s IP adresami patřící do seznamu zakázaných adres. Tento seznam může obsahovat velké množství položek. V pravidlu by proto měla být možnost definování proměnné, která by na blacklist odkazovala.

6. *Příliš mnoho toků obsahujících pouze* **jeden TCP flag**.
   Záplava paketů s příznakem SYN nacházející se v hlavičce TCP způsobuje taktéž vyčerpání zdrojů serveru stejně jako v případě útoku typu SIP flood. Podobně jako v případě detekce reflektorů i zde může existovat více přístupů, jak identifikovat příliš mnoho toků. Například se může hlídat překročení určitého procenta toků obsahující příznak SYN z celkového množství toků pro danou IP adresu.

7. *Na žádnou IP adresu z podsítě nesmí jít více než určité množství určitého (např. ICMP)* **provozu za sekundu** *(bps/fps), to stejné i pro opačný směr.*
   V síťových tocích obvykle bývá informace o typu transportního protokolu, protože bývá součástí pětice klíče toku, takže by nebylo problém zmíněný protokol ICMP identifikovat. Počítání množství přenesených bytů za sekundu (bps) by se dalo odvodit ze síťového toku prostřednictvím údajů popisujících dobu trvání toku a počtu přenesených bytů.

Z rozboru jednotlivých potencionálních potřeb správce vyplývá, že společným prvkem každého případu je v prvé řadě filtrace. Toto zjištění není tolik překvapivé, protože každá detekce je vlastně úzce zaměřena na nějaký problém, tudíž potřebuje prostředek, jak extrahovat ze síťových toků pouze potřebnou část. Kromě běžných filtračních úloh by měla být filtrace přizpůsobena pro identifikování adres podsítě či k filtrování na základě příznaků, jenž mohou být reprezentovány v binárním formátu. Dalším častým společným prvkem je nějaká agregační funkce, která např. počítá součet komunikačních spojení či příkazů pro každou IP adresu. Vypočítaná hodnota se poté porovná s nějakou definovanou proměnnou označovanou jako X a pokud je větší než tato proměnná, tak má dojít k nahlášení porušení bezpečnostní politiky.

4.3 Zhodnocení zkoumaných systémů

V kapitole 3 byly v sekcích 3.3 a 3.4 představeny systémy zaměřené na proudové zpracování dat. V předchozí sekci z analýzy vyplynulo, že správce by si mohl vystačit s první filtrací, agregací pro zvolený časový interval a druhou filtrací. Pravidlo by mělo minimálně poskytovat tyto funkce a navíc by měly také existovat možnosti pro uspokojení dalších nespecifikovaných potřeb správce. Než dojde ke shrnutí zkoumaných systémů založených na jazyce SQL, nechť je ještě předtím probrána obecná struktura dotazu:

```
SELECT < sloupce >
FROM < zdroje dat >
WHERE < predikáty >
GROUP BY < sloupce >
```

```
HAVING < predikáty nad skupinami >
ORDER BY < sloupce >
```

První zmíněná filtrace se nachází v klauzuli `WHERE`, kde jsou jednotlivé predikáty od sebe odděleny binárními operátory konjunkce a disjunkce. V klauzuli je také prostor pro přidání poddotazu, čímž roste složitost celého dotazu. Tvoření skupin probíhá podle klíče v klauzuli `GROUP BY`. Agregační funkce se poté nachází v klauzuli `SELECT` či `HAVING`, jejichž parametrem je vybraný sloupec. Druhé filtrování probíhá nad skupinami v klauzuli `HAVING`, kde se zadává agregační funkce, jejichž výsledek je porovnán s relačním operátorem a definovanou hodnotou. Jak bylo probráno v kapitole 3, jazyk SQL se musí rozšířit o konstrukci posuvného okna, aby bylo možné tento jazyk použít pro dotazy nad proudem dat. Potom lze konstatovat, že minimální operace, které by správce sítě potřeboval, jsou v rozšířeném SQL dotazu obsaženy. Nyní následuje shrnutí probraných systémů.

STREAM nepodporuje ve své poslední funkční verzi klauzuli `HAVING`, čímž je porušena základní podmínka pro splnění potřeb správce, protože chybí druhá fáze filtrování. Sepsané práce a články projektu STREAM obsahují nicméně užitečné informace, které se mohou využít při implementaci modulu této diplomové práce. Například zmiňované řešení problému redundance paměti pomocí konstrukcí stub a storage.

TelegraphCQ nepodporuje operátor disjunkce v klauzuli `WHERE`. První fáze filtrování probíhající v této klauzuli se tedy komplikuje. Myšlenka využívání relační databáze v dotazech by mohla být zajímavým rozšířením výsledného modulu, nicméně implementace by byla nejspíš uskutečněna v některých dalších verzích, pokud by to bylo potřeba. V první verzi modulu podpora není zdaleka nutná.

WSO2 představuje zástupce moderních proudových systémů, které poskytují služby a funkce daleko nad rámec řešení potřeb správce, což je velmi dobré. Otázkou je, jak by nakonec probíhala spolupráce systémů NEMEA se systémem jako je WSO2, jakmile by byla přidána podpora pro zpracování síťových toků. Na místě je také obava, jestli by nakonec tyto proudové systémy uspokojovaly časové a paměťové omezení, jelikož většina z nich běží nad virtuálním strojem Java (Java Virtual Machine, JVM). Po konzultacích o směru této diplomové práce bylo však rozhodnuto, že zkoumání těchto otázek není předmětem této práce.

Stream4Flow poskytuje pohled na to, jak by takové dotazování nad síťovými toky v moderním proudovém systému vypadalo. Používané moduly, v nichž jsou dotazy popsány, na první pohled nevypadají zrovna jednoduše, ale na druhou stranu programátorovi pravděpodobně poskytují dostatek volnosti pro řešení komplexnějších úloh.

NFQL na první pohled poskytuje potřebné operace pro dosažení cíle. Stádium Filter je sémanticky podobné klauzuli `WHERE` v jazyce SQL, stádium Grouper poskytuje stejnou funkci jako klauzule `GROUP BY`, ale navíc se v tomto stádiu řeší agregační funkce. Druhá fáze filtrování je zajištěná stádiem Group-filter, která ve své podstatě dělá stejnou činnost jako klauzule `HAVING`. V systému NFQL chybí pouze definování posuvného okna, jinak jsou základní požadavky splněny. Stádia Merger a Ungrouper jsou tedy navíc a jejich přítomnost není pro potřeby správce nutná. Analogicky je stádium Merger užitečné jako operátor `JOIN` v klauzuli `FROM` s tím rozdílem, že jeho funkce je ve spojování skupin do n-tic. Stádium Ungrouper žádné přirovnání s klauzulí nemá, ale možná by šla jeho funkce v jazyce SQL napodobit nějakým složitějším dotazem.

Co se týče nasazení systému, tak jak bylo zmíněno v sekci 3.4, implementace obsahuje spoustu nevyřešených problémů a není jasné do jaké míry by bylo třeba upravit zdrojové kódy, aby byl systém schopen zpracovávat data proudovým způsobem.

Z rozboru plyne, že jazyk NFQL se nejvíce hodí pro splnění zadání této diplomové práce. Jazyk splňuje základní požadavky pro uspokojení stanovených potřeb a navíc svou jednoduchostí umožňuje rychlé pochopení toho, jak zpracování dat probíhá. Část jazyka by se ale měla upravit nebo případně rozšířit o nové užitečné funkce.

4.4 Navržená struktura jazyka

Navržený jazyk vychází z jazyka NFQL, jenž je okomentovaný v sekcích 3.4 a 4.3. Jazyk je však potřeba upravit, jelikož některé jeho konstrukce jsou nadbytečné nebo dokonce chybí.

Konkrétně stádia Merger a Ungrouper se jeví jako zbytečná, protože pro řešení potřeb správce 4.2 jich není třeba. Navíc stádium Merger vykazuje vysokou časovou složitost $O(m^n)$, kde n je počet větví poskytující skupiny a m je počet těchto skupin. Stádium Ungrouper se jeví nevhodné z hlediska paměťové náročnosti, protože při jeho použití se musí uchovávat všechny síťové toky v paměti, dokud si je Ungrouper nevyžádá. Ukládat se tedy musí agregační hodnoty i síťové toky po stanovenou dobu, což by mohlo výrazně snížit rychlost zpracování dat celého systému.

V jazyku NFQL chybí možnost definovat posuvná okna, přitom jejich uživání přináší výhody 3.2. Jazyková konstrukce pro posuvná okna by měla být dostatečným prostředkem pro popis časových podmínek u zpracování, jelikož jsou běžnou součástí proudových systémů. Allen's time interval algebra používaná v jazyku NFQL je tedy nahrazena posuvnými okny.

Navrhovaný jazyk poskytuje uživateli popis proudového zpracování dat pomocí stromové struktury. Kořenem stromu je stádium Splitter, který distribuuje příchozí data do větví. Větev se může postupně skládat ze stádií Filter, Grouper, Window, Aggregator, Groupfilter, Selector či Publisher, anebo se může větev rozdělit do dalších větví. Před rozdělením větví se implicitně vkládá stádium Splitter, takže ho uživatel vůbec nepíše. Pro stádium platí, že se nesmí v cestě od kořene stromu po jeho list opakovat. Tato cesta zároveň znázorňuje proudové zpracování. Stádium se skládá ze svého názvu a těla, které je pro každý typ stádia specifické. Následuje popis jednotlivých stádií:

Filter propouští síťový tok do dalších stádií, pokud se zadaná podmínka vyhodnotí jako pravdivá. Syntaxe filtru je inspirována z bakalářské práce [23]. Podmínka se skládá z výrazů, jenž jsou odděleny logickými operátory. Užitečná je také operace *in* zejména ve spojení s IP adresami či porty. Operace slouží pro porovnání hodnot ze seznamu. To se dá využít např. v případě definování podsítí. Další operace *exist* ověřuje přítomnost atributu v síťovém toku.

Grouper seskupuje síťové toky podle definovaného klíče. Klíč se skládá z jednoho či více atributů, typicky z IP adresy.

Window definuje posuvné okno pomocí parametrů `WATTR`, `RANGE` a `SLIDE`. Parametrem `WATTR` se určuje, který atribut v síťovém toku má plnit roli časové značky. Parametrem `RANGE` se definuje velikost posuvného okna a volitelný parametr `SLIDE` značí, o kolik časových jednotek se má okno posunout. Jestliže `SLIDE` není definovaný, posunuje se

okno o velikost definovanou v `RANGE` a okno se tedy chová jako Tumbling window. V opačném případě se okno chová jako Hopping window.

Aggregator je stádium, v němž se definují jména proměnných, do kterých se přiřadí výsledek agregačních funkcích. Identifikátor proměnné se dále využije zejména ve stádiu Group-filter či Selector. Agregační funkce mohou být `COUNT` pro počítání položek ve skupině, `SUM` pro sčítání hodnot, `AVG` pro průměr a další. Výstupem stádia Aggregator je tedy skupina skládající se z proměnných a klíče.

Group-filter filtruje skupiny podle proměnných definovaných ve stádii Aggregator. Podmínky mají stejnou syntaktickou strukturu jako v případě stádia Filter.

Selector má za úkol sestavit hlášení. Parametr stádia Selector může být dvojího druhu. Buď se napíše identifikátor proměnné či jméno atributu síťového toku, anebo se přidá popis pomocí textového řetězce. V tom případě lze navíc hodnotu z proměnné či atributu upravit pomocí aritmetického výrazu.

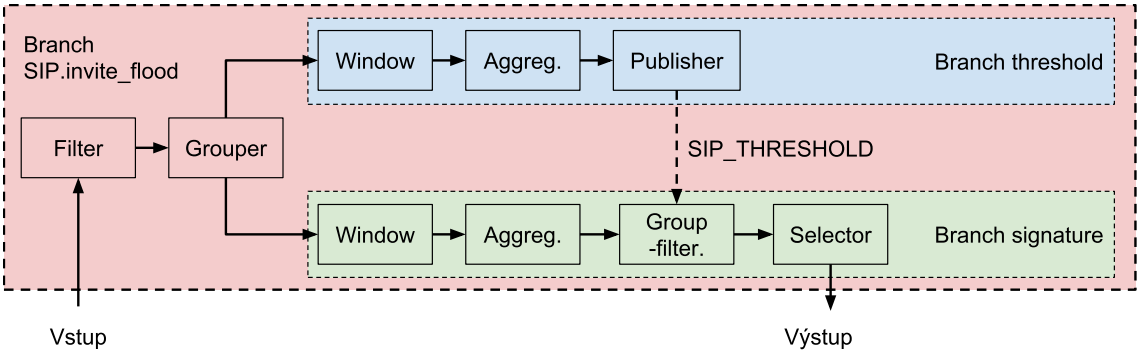
Publisher zpřístupňuje výsledky jedné větve ostatním větvím. Výsledek se ukládá do definované proměnné, která je přístupná ve stádiích Filter, Group-filter a Selector. Stádium Publisher má výhradní právo do proměnné zapisovat novou hodnotu, zatímco ostatní stádia mají pouze právo ke čtení. Stejně jako v případě stádia Selector může být výsledná hodnota v proměnné vypočítána na základě aritmetického výrazu. Význam stádia je ve sledování statistických vlastností síťových toků. Zjištěné hodnoty by se poté použily zejména při filtrování.

Následuje ukázková demonstrace jazyka doplněná dodatečným popisem a obrázkem stromové struktury 4.1:

```
branch SIP {
  branch invite_flood {
    filter:
      DST_IP in $SIP_SERVERS and (SIP_MSG_TYPE == 5 or SIP_MSG_TYPE == 1);
    grouper: SRC_IP, DST_IP;

    branch threshold {
      window: WATTR = TIME_FIRST, RANGE = 300 seconds, SLIDE = 60 seconds;
      aggregator: connection_count = COUNT(*);
      publisher: SIP_THRESHOLD = connection_count * 0.4;
    }
    branch signature {
      window: WATTR = TIME_FIRST, RANGE = 60 seconds, SLIDE = 10 seconds;
      aggregator: connection_count = COUNT(*);
      group-filter: connection_count > SIP_THRESHOLD;
      selector: attacker = SRC_IP, victim = DST_IP;
    }
  }
}
```

- Větev *SIP* je specifikována následující větví `invite_flood`. Ve zpracování dat tato následnost nemá žádný vliv, ale slouží to pouze pro přehlednější organizaci jmen



Obrázek 4.1: Stromová struktura proudového zpracování dat.

větví. V ukázce jsou fakticky dvě větve, kterým probíhá zpracování dat. Jejich zápis v tečkové notaci je: `SIP.invite_flood.threshold` a `SIP.invite_flood.signature`.

- Pomocí znaku $ lze odkazovat na proměnnou v konfiguračním souboru, kde je proměnná definovaná. Díky proměnné se v uvedeném případě nezahltí zápis IP adresami.
- V ukázce jsou stádia Filter a Grouper společná pro větve *threshold* a *signature*. Větev *threshold* zprostředkovává globální proměnnou `SIP_THRESHOLD`, která nabízí díky většímu posuvnému oknu širší pohled na síťové toky a zároveň se zde definuje míra meze. Větev *signature* využívá tuto globální proměnnou a hlídá překročení meze.
- Znak hvězdička jako parametr v agregační funkci `COUNT` znamená, že se počítá počet síťových toků. Výsledek agregační funkce se uchovává v proměnné *connection_count*. Její rozsah působnosti platí ve větvi, ve které je definovaná a platil by dál i v navazujících větveních.
- Systém musí pouze uchovávat agregační hodnoty do doby, než se vypočítá první hodnota v `SIP_THRESHOLD`, což je po 60 sekundách. To znamená, že systém do té doby Group-filter nepoužívá a nemohou se tedy produkovat hlášení.
- V konfiguračním souboru by mohla být také možnost definování nečinnosti větve po určitou dobu startu systému, zatímco ostatní větve normálně zpracovávají data se startem systému. To má smysl ve výše uvedené ukázce, kdy hrozí, že útok nastane např. po 590 sekundě, čímž se v 600 sekundě do proměnné `GLOBAL_SIP_THRESHOLD` přiřadí hodnota obsahující zachycení útoku. Podmínka ve stádiu Group-filter se tudíž nemusí splnit, i když by měla. Pokud se ale start větve *threshold* posune např. o 5 vteřin, tak se v 600 sekundě podmínka v Group-filter splní, protože proměnná `GLOBAL_SIP_THRESHOLD` nebude zatím aktualizovaná. Další řešení by mohlo být, že se proměnné ve stádiu Publisher aktualizují až po vyhodnocení podmínek ve filtrech.

Výsledkem této kapitoly je návrh jazyka, pomocí něhož by měl být uživatel schopen napsat vlastní pravidla odhalující nežádoucí provoz v síti. Výslednému návrhu nejprve předcházel komentář k části teoretické kapitoly. Nejdůležitější bylo zjištění, že monitorovací sonda má více možností, jak zachytávat síťový provoz. Může tedy nastat situace, že záznamy toků od různých sond mají odlišnou charakteristiku zejména kvůli rozdílné strategii při exportování záznamů směrem do kolektoru. Dokonce mohou některé záznamy chybět,

tudíž modul může na vstupu dostávat nekompletní data a je proto třeba mít na vědomí, že tyto příčiny mohou negativně ovlivnit detekování nežádoucího provozu.

Dále se v této kapitole zkoumaly potenciální potřeby správce. Bylo možné v případech rozpoznat opakující se operace filtrování a agregování s následným filtrováním. Práce poté pokračovala zkoumáním toho, jak by bylo možné tyto tři operace přehledně zapsat v nějakém jazyce. Obvykle se pro dotazování nad proudem dat používá v současných systémech rozšířená varianta SQL. Detailní náhled na jazykové konstrukce ukázal, že tento jazyk požadavkům vyhovuje. Při zkoumaní dostupných systémů se však vývojáři potýkali s obtížemi, které nakonec vyústily nekompletními implementacemi. Navíc se zdá být rozšířený jazyk SQL příliš komplexní v kontextu této diplomové práce. Řešení se nakonec nabízelo v jazyce NFQL. Autoři přinesli nový pohled na formu zápisu a přišli také s myšlenkou větvení. Bylo ale nutné jazyk patřičně upravit a nejlépe také ještě více zjednodušit.

Mé navržené úpravy jazyka NFQL zahrnovaly odebrání zbytečných stádií Splitter, Merger a Ungrouper. Naopak jsem přidal stádium Window, ve kterém je možné definovat časové okno, a také jsem přidal stádium Selector umožňující sestavit výstupní položky. Navíc jsem také navrhl stádium Publisher, který má poskytnout relativní charakteristiky vstupních dat ostatním větvím. Větvení samotné jsem přepracoval do jiné podoby, aby umožňovalo přehlednější pohled na dělení zřetězeného zpracování. Zároveň díky novému přístupu větvení má uživatel možnost optimalizovat svá pravidla, a sloučit tak pravidla obsahující stejná počáteční stádia. Celkově jsem také zjednodušil těla parametrů u stádií. Největší změnou parametrů prošlo stádium Aggregator, ve kterém je nově možné výsledek agregační funkce přiřadit do proměnné, kterou je dále možné využívat v následujících stádiích. To přispívá k lepšímu intuitivního pochopení fungování pravidla.

Kapitola 5

Implementace

Tato kapitola se zabývá implementací modulu **Policer** určeného pro systém NEMEA. První část kapitoly se zabývá rozhraním modulu 5.1 a příkladům možného použití. Dále kapitola pokračuje popisem architektury modulu z pohledu nejvyšší abstrakce 5.2. V druhé části této kapitoly je podrobněji rozepsáno fungování jednotlivých dílčích klíčových částí architektury. Nejprve je vysvětleno, jakým způsobem je řešena syntaktická a sémantická analýza navrženého jazyka 5.3. Dále je rozebrána implementace dvou důležitých komponent zabývající se filtrováním 5.4 a agregováním 5.5 vstupních dat. Tyto komponenty byly převzaty z jiných závěrečných prací a přizpůsobeny vnitřnímu prostředí modulu. V navazující sekci 5.6 je poté vysvětlena potřebná substituce identifikátorů. Kapitola je zakončena shrnutím, čím se výsledná implementace liší od stanoveného návrhu.

5.1 Rozhraní modulu

Modul Policer je provázán s komunikačním rozhraním knihovny **LibTrap**. Základní vlastnosti této knihovny byly v této práci již zmíněny v sekci 2.4. Knihovna tedy poskytuje komunikační rozhraní s ostatními moduly systému NEMEA a díky jejich vzájemnému propojení do podoby acyklického grafu lze sestavit komplexní nástroj pro analyzování síťových toků. Příkladem jednoduchého avšak užitečného propojení modulů může být kombinace *logreplay-policer-logger*. Modul *logreplay* je schopen načíst data ve formátu CSV a exportovat záznamy síťových toků dále v podobě formátu UniRec. Modul *logger* naopak dělá zpětnou konverzi a je schopen uložit příchozí data do formátu CSV. Díky tomuto jednoduchému propojení může každý modul, včetně modulu Policer, číst a zapisovat síťové toky ve formátu CSV.

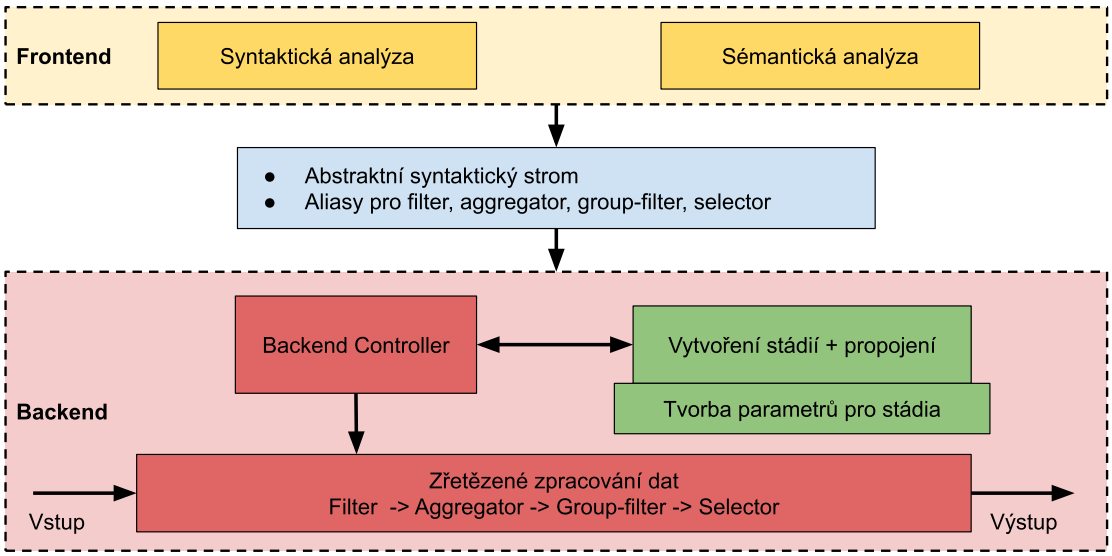
Podpora LibTrap knihovny přináší mimo jiné ještě jednu důležitou funkcionalitu a tím je rozhraní pro čtení a zápis souboru ve formátu **.trapcap**. Tento formát umožňuje uchovávat data síťových toků v binární podobě, takže dochází k menšímu zaplnění paměťového místa. V kontextu modulu Policer může být tato funkcionalita velmi užitečná, protože pokud by uživatel zadal více pravidel pro odhalování nežádoucího provozu, tak by výstup každého pravidla mohl být uložen ve zvláštním binárním souboru.

Díky knihovně LibTrap se rozhraní řeší před spuštěním modulu na příkazové řádce. Z implementačního hlediska se v modulu Policer vůbec neřeší odkud data přicházejí, či jaký má být výstupní formát. Knihovna v nejjednodušší variantě nabízí potřebnou míru abstrakce, stačí tedy zavolat funkci pro získání dat a přeposlání dat dále.

Rozhraní modulu Policer by se dalo rozdělit do dvou částí. V první části se nachází parametry pro knihovnu LibTrap a v druhé části se zadává cesta k souboru, v němž se nachází uživatelem definovaná pravidla. U parametru pro LibTrap knihovnu je pevně stanoveno, že vstupní rozhraní, ve kterém mají proudit záznamy síťových toků, musí být právě jedno. Výstupních rozhraní může být vícero, to záleží především na počtu Selector stádií v textovém souboru s pravidly. Obdobně by se dalo také říci, že počet Selector stádií udává, kolik pravidel pro odhalení nežádoucího síťového provozu se v souboru nachází. Maximální počet pravidel je ale limitován samotnou LibTrap knihovnou, protože knihovna nedovoluje vytvořit více než 32 výstupních rozhraní u jednoho modulu.

5.2 Architektura

Schéma architektury modulu Policer je znázorněno na obrázku 5.1. Schéma se dělí na dvě hlavní části, které nesou jméno Frontend (horní část obrázku) a Backend (dolní část obrázku). V první části je patrné, že se řeší syntaktická a sémantická analýza jazyka, která je podrobněji rozebrána v sekci 5.3. Výstupem této části je datová struktura obsahující vnitřní reprezentaci zpracovaného jazyka. Ta se skládá z abstraktního syntaktického stromu získaného syntaktickou analýzou a z tzv. aliasů, viz dále.

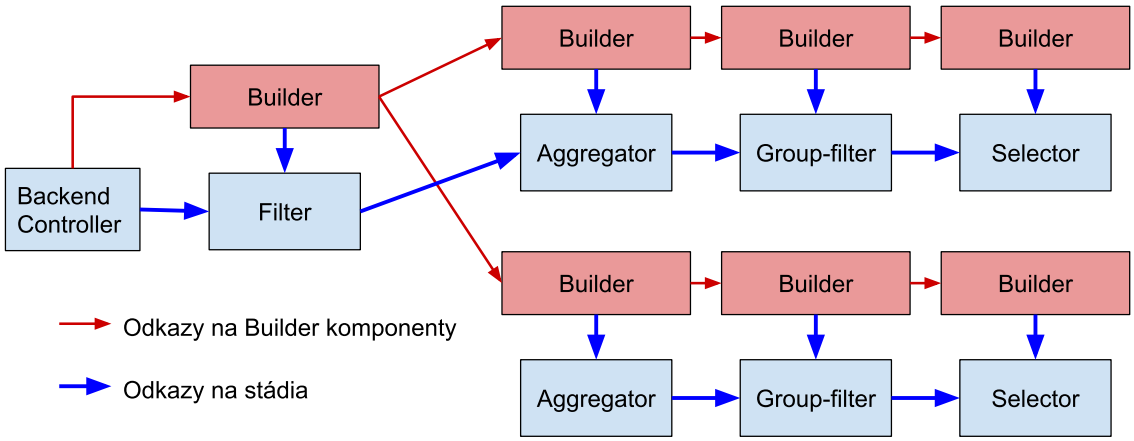


Obrázek 5.1: Architektura modulu Policer.

Vnitřní reprezentace je poté předána části s názvem Backend, kterou řídí jednotka Backend Controller. Ze syntaktického stromu se poté odvodí jakou strukturu bude mít zřetězené zpracování, tedy z jakých stádií se bude zpracování skládat a jak budou jednotlivá stádia propojena. U propojování se bere především v úvahu, kteří přímí následníci jsou umístěni za sledovaným stádiem. Pokud se tedy za stádiem nachází více než jedno stádium, dochází k větvení zřetězeného zpracování a výstupní data sledovaného stádia musí být předána všem přímým následníkům.

Na obrázku 5.2 je již zmíněné provázání jednotlivých stádií znázorněné modrými šipkami. Každé stádium tedy má k dispozici odkazy na svoje následníky. Na obrázku jsou dále znázorněny komponenty Builder, které mají na starost konstrukci a destrukci svých přidělených stádií. Stádia totiž sama o sobě mají jistou úroveň složitosti, tudíž by nebylo

vhodné jim přidělit ještě navíc odpovědnost za destrukci (dealokaci) svých následníků. Tato zodpovědnost je tedy přenechána komponentám Builder, které jsou provázané odkazy (červenými šipkami) obdobně jako jejich spravovaná stádia. Pořadí alokování probíhá od posledního stádia a končí stádiem prvním, protože se tímto způsobem snadno předávají odkazy na následníky. Naopak destrukce probíhá od prvního stádia a končí destrukcí posledního, jelikož např. v agregátoru mohou být ještě v tu dobu přítomna neodeslaná data, tudíž se nejprve data odešlou a až potom se může zavolat jeho destruktor.



Obrázek 5.2: Schéma sestavení komponent.

S vytvářením stádií je také spojené sestavení příslušných parametrů. Parametry jsou pro každý typ stádia různé a jejich konečná podoba je vyhodnocena na základě uložených informací z abstraktního syntaktického stromu. Důležitou procedurou při sestavování parametrů je změna jmen některých proměnných na odpovídající klíčová slova z UniRecu. K tomu se právě využívají aliasy, jenž jsou podrobněji rozepsány v sekci 5.6.

Jakmile jsou stádia řádně sestavena a nedošlo k žádné chybě, může komponenta Backend Controller zahájit zřetězené zpracování síťových toků UniRec záznamů. Knihovna LibTrap zajistí na vyžádání přísun nových záznamů, které jsou postupně rozeslány do jednotlivých větví ke zpracování. V modulu Policer se spouští více vláken, ale při detailnější analýze vychází najevo, že prakticky modul většinu času pracuje jako jedno-vláknová aplikace. Detailněji se tímto problémem zabývá sekce o aggregátoru 5.5, protože právě v tomto stádiu je při nastavení určitých parametrů nutné mít v běhu vlákno kontrolující časové okno.

Uživatel má možnost pravidlo odhalující nežádoucí provoz zapsat nejvýše pomocí šesti stádií. V Backend části se ale používají celkem tři stádia — Filter, Aggregator a Selector. Stádia Grouper a Window jsou totiž v podstatě jen parametry patřící pod agregaci. Na obrázku 5.3 je znázorněno jejich společné rozhraní v jazyce UML.

5.3 Syntaktická a sémantická analýza

Pro syntaktickou a sémantickou analýzu byla zvolena knihovna Boost Spirit X3 [11]. Knihovna nabízí objektově orientovaný přístup při používání rekurzivního syntaktického analyzátoru, jenž je založen na gramatice PEG (Parsing Expression Grammars) [10]. V této formální gramatice lze zapisovat přepisovací pravidla gramatiky v podobném formátu jako u Rozvinuté Backus-Naurovy formy (EBNF) [30], ale figuruje zde odlišná interpretace pravidel. Nejdůležitější rozdíl je u výběrového operátoru (značí se symbolem /, případně |). U něj

Obrázek 5.3: Společné rozhraní pro stádia Filter, Aggregator, Selector

≪interface≫ **Stage_interface**
#pipeline_successors : vector<Stage_interface*>
+init(options : char const, succ : const vector<Stage_interface*>) : int* *+eval(rec : void const*, in_tmplt ur_template_t const*) : int* #send(out_rec : void const*, out_tmplt : ur_template_t const*) : void *+~Stage_interface()*

platí, že se vždy vybírají přepisovací alternativy v pevném pořadí směrem zleva doprava. Pro PEG dále platí, že je bezkontextovou gramatikou (context-free, CFG) neobsahující cykly, tudíž není víceznačná a také nesmí obsahovat pravidlo, jenž by bylo rekurzivní zleva.

Přepisovací pravidla gramatiky se zapisují přímo v C++ kódu a díky možnosti přetěžovat C++ šablony je pravidlo plně integrované v uživatelském programu. Pokud je zápis validní, je možné kód přeložit klasickým způsobem pomocí kompilátoru *g++*. Nevolá se tedy externí program, jenž by pravidla gramatiky zkompiloval a vygeneroval kód v C/C++, jak je tomu zvykem například u nástrojů Flex a Bison [1]. V následující ukázce je demonstrováno porovnání formálního zápisu klasického přepisovacího pravidla v PEG se zápisem v Spirit X3:

```
PEG přepisovací pravidlo:
aggregator_body <- aggr_assign (',' aggr_assign)*

Totéž přepisovací pravidlo vyjádřené pomocí Spirit X3:
auto const aggregator_body_def = aggr_assign > *(',' > aggr_assign);
```

Z ukázky je patrné, jak se autoři knihovny snažili přiblížit formálnímu zápisu. V rámci této diplomové práce bylo třeba nejprve formálně zapsat gramatiku jazyka v PEG, viz příloha A. V dalším kroku bylo třeba pravidla implementovat, což bylo provedeno přímočarým způsobem díky výrazné podobnosti formátu pravidel.

Kromě samotných přepisovacích pravidel bylo dále nutné definovat abstraktní syntaktický strom, opět pouze pomocí jazyka C++. Toho se docílí pomocí deklarace struktur, kde každá struktura může odpovídat jednomu pravidlu. Lze si představit, že po vyhodnocení pravidla se do struktury uloží odpovídající data. Pokud pravidlo využívá výběrového operátoru, lze tuto skutečnost vyjádřit ve struktuře pomocí konstrukce *x3::variant*[1]. Abstraktní syntaktický strom je tedy složen z několika do sebe zanořených struktur, ať už přímo nebo s pomocí konstrukce *variant*.

Pro vytvoření abstraktního syntaktického stromu bylo tedy nutné v modulu Policer definovat přepisovací pravidla gramatiky a složenou strukturu představující syntaktický strom. Pro následnou sémantickou analýzu bylo ovšem potřeba syntaktický strom procházet a požadované položky zkontrolovat, zda uživatelem definované pravidlo odhalující nežádoucí provoz dává smysl. Pro procházení syntaktického stromu se v knihovně Spirit X3 používá rekurzivní funkce. Rekurzivní funkce má podobu **void operator()**, kde parametrem je datový typ dílčí struktury. Díky možnosti přetěžování operátoru lze definovat akci pro každou dílčí strukturu. Postupně se tedy dá zanořovat pomocí rekurzivního volání do libovolné částí syntaktického stromu.

[1] https://theboostcpplibraries.com/boost.variant

Pro nějakou sémantickou kontrolu jsou tedy připraveny třídy s definicemi vybraných rekurzivních funkcí, ve kterých jsou definované akce v podobě sémantické kontroly. Celkem byly implementovány tyto sémantické kontroly, které ověřují:

- validní posloupnosti stádií ze seznamu povolených posloupností,
- unikátní jméno větve v lokální rámci působnosti,
- nepřítomnost zakázaných identifikátorů ve stádiu Grouper,
- zákaz přítomnosti stejného identifikátoru ve stádiu Grouper a jeho navazujícím stádiu Aggregator,
- validní přítomnost identifikátoru ve stádiích Filter, Group-filter a Selector,
- ověření existence identifikátoru v příslušném Aggregator stádiu.

Některé sémantické kontroly týkající se stádií Grouper a Aggregator implicitně nevyplývají z navrženého jazyka. Kontroly musely být dodatečně vytvořeny na základě limitujícího použití implementovaného agregátoru, viz sekce 5.5.

5.4 Filter

Implementace filtru byla převzata z bakalářské práce [23] a zdrojový kód je volně dostupný z Github repozitáře[2]. Tato implementace filtru byla zvolena na základě doporučení a navíc se filtr reálně používá např. v rámci projektu libnf[3]. Syntaxe pro zápis filtrovacího výrazu byla inspirována nástrojem nfdump. Díky flexibilnímu rozhraní je možné filtr přizpůsobit konkrétní aplikaci. Uživatel pouze předá informaci o datovém typu identifikátoru nacházejícího se ve výrazu. Také dále předá k identifikátoru callback funkci, jejíž úkolem je předat filtru data pro vyhodnocení výrazu. Filtr pouze aplikuje nad získanými daty svoji vnitřní logiku, která je odvozená ze zadaného filtrovacího výrazu. Poté následně uživateli vrátí informaci o pravdivosti výrazu. Aby byl filtr dostatečně výkonný, je kompletně napsaný v jazyce C, ale ničemu nebrání mít callback funkce napsané v jazyce C++.

V rámci implementace modulu Policer se filtr používá v řetězeném zpracování síťových toků ve dvou stádiích — Filter a Group-filter. Ve stádiu Group-filter se ale používá jiný typ identifikátorů. Zatímco ve stádiu Filter jména identifikátorů pocházejí z formátu UniRec, tak ve stádiu Group-filtr se nacházejí navíc identifikátory proměnných definovaných ze stádia Aggregator. K tomu se váže problém, jak připravit filtr na tyto uživatelem napsané identifikátory, které mohou být z libovolného řetězce znaků. Tento problém je v modulu Policer vyřešen interním přejmenováním uživatelem definovaných identifikátorů na odpovídající UniRec identifikátory (aliasy). Bližším vysvětlením se zabývá sekce 5.6.

Zdrojový kód filtru se také používá v části zabývající se sémantickou analýzou. V rámci sémantických kontrol byla totiž potřeba zjistit, zda jsou validní zadaná jména identifikátorů ve filtrovacím výrazu. Zdrojové kódy filtru byly tedy upraveny tak, aby mohl filtr vrátit seznam používaných identifikátorů, nad kterými poté probíhaly příslušné sémantické kontroly.

Filtr také umožňuje ve filtrovacím výrazu používat např. klíčová slova *src* a *dst* ve spojení se slovem *ip*. Vyjádření směru umožňuje pohodlnější a v některých případech kratší

[2] https://github.com/VUTBR/libnf-ffilter
[3] https://github.com/VUTBR/libnf

zápis výrazu. Tato funkcionalita není v modulu Policer zatím podporována, nicméně je vždy možné vymyslet ekvivalentní alternativu, která ale bude vyžadovat delší zápis výrazu.

Filtr vyhodnocuje na základě derivačního stromu, jenž vznikne po překladu výrazu, takže se nad výrazem neaplikují žádné optimalizace. Z tohoto důvodu není tento filtr vhodný např. pro ověření, jestli se IP adresa nachází v seznamu nežádoucích adres, protože filtr v podstatě prochází seznam lineárně. Tento filtr je tedy efektivní pouze pro kratší výrazy. Na druhou stranu nutno konstatovat, že v budoucnu se pro filtr plánuje podpora pro vyhledávání v rozsáhlém seznamu IP adres.

V rámci této diplomové práce bylo potřeba doplnit uživatelské callback funkce. První funkce *lookup_func* se aplikuje při inicializaci filtru a seznamuje jej se všemi identifikátory UniRec, kde se k nim nastaví odpovídající datové typy. Druhá doplněná funkce *data_func* se poté využívá během běhu modulu a má za úkol vytáhnout ze vstupního UniRec záznamu odpovídající data a předat je filtru. Aby bylo toto možné, bylo nutné lehce upravit rozhraní filtru pro zpřístupnění UniRec šablony, která pomáhá ze záznamu žádaná data extrahovat. Při tomto úkolu se objevil problém s rozdílnou bitovou reprezentací IPv4 adresy v UniRecu a filtru. Ve funkci *lookup_func* musela být kvůli tomuto doplněna bitová konverze, aby mohl filtr fungovat správně.

5.5 Aggregator

Implementace agregátoru byla převzata z diplomové práce [22], jejíž zdrojové kódy jsou volně dostupné v Github repozitáři[4]. Agregátor využívá datový formát UniRec a je plně integrován ve formě modulu v rámci systému NEMEA. Modul je implementovaný v jazyce C++ a jeho konceptuální návrh podporuje možnost rozšíření o další agregační funkce bez nutnosti zásahu do jádra zdrojového kódu modulu. Výstupem agregátoru je záznam, u kterého pro všechny položky platí, že výsledek agregační funkce se ukládá ke stejnému identifikátoru, ze kterého se výsledek spočítal. Z toho vyplývá omezení pro používání agregátoru takové, že nad konkrétním identifikátorem není možné aplikovat více jak jednu agregační funkci.

Dostupné agregační funkce umožňují výpočet sumy, průměru, nejmenšího a největšího prvku, prvního a posledního prvku, operace bitového součtu a součinu. Implicitně se také počítá celkový počet agregovaných záznamů. V rámci této diplomové práce bylo potřeba přidat dvě agregační funkce, kde první počítá sumu prvků za vteřinu (`RATE`) a druhá funkce počítá počet unikátních prvků (`COUNT_DISTINCT`). Funkci `RATE` bylo možné bez větších problémů implementovat, díky připravenosti návrhu na tento potencionální požadavek. U implementace funkce `COUNT_DISTINCT` byl ale nutný větší zásah. Důvodem je nová potřeba někde v paměti ukládat dosavadní prvky, aby bylo poté možné informovat o počtu unikátních prvků. Řešení je naznačeno na obrázku 5.4.

Na obrázku 5.4 lze vidět příklad jednoho spravovaného záznamu hašovací tabulky, kde klíčem pro agregování je zdrojová IP adresa (`SRC_IP`). Na obrázku je znázorněno, že položka s identifikátorem `COUNT_DISTINCT` neobsahuje hodnotu, ale ukazatel do paměti, kde se shromažďují unikátní cílové IP adresy (`DST_IP`). Položky se uchovávají ve standardním C++ vektoru a vyhledávání je implementované lineárním procházením. Ve chvíli, kdy je potřeba záznam exportovat, tak se musí spočítat počet unikátních položek. Výsledný exportovaný záznam je nakonec znázorněn na obrázku 5.5

[4] https://github.com/CESNET/Nemea-Modules/tree/master/aggregator

SRC_IP	COUNT_DISTINCT_DST_IP	PACKETS	COUNT	UniRec identifikátory
192.168.0.2	ukazatel	4621	8	Hodnoty

192.168.48.39	192.168.72.14	192.168.153.5	Vektor unikátních hodnot

Obrázek 5.4: Schéma řešení agregační funkce `COUNT_DISTINCT`.

SRC_IP	COUNT_DISTINCT_DST_IP	PACKETS	COUNT	UniRec identifikátory
192.168.0.2	3	4621	8	Hodnoty

Obrázek 5.5: Schéma exportovaného záznamu.

Agregátor odesílá záznam na základě nastaveného typu timeoutu. V režimu Aktivního timeoutu se každý příchozí záznam porovnává s příslušným uloženým záznamem. K odeslání dojde v okamžiku, kdy hodnota `TIME_FIRST` u příchozího záznamu je větší než součet `TIME_FIRST` hodnoty v uloženém záznamu a hodnoty parametru `-t`. V režimu Pasivního timeoutu se procházejí v časových intervalech všechny uložené záznamy. Pokud se záznam po definovanou dobu neaktualizoval, je přeposlán na výstup. Akivní a Pasivní režimy jsou podobné časovému oknu Session window, jenž byl zmíněn v sekci 3.2. Časové okno Tumbling window lze v agregátoru získat pomocí Globálního timeoutu. Na výběr je také režim Mix, který je kombinací Aktivního a Pasivního režimu.

Při nastavení Pasivního nebo Globálního timeoutu pracují v modulu dvě vlákna řízení. První vlákno má na starost vkládání nového záznamu do hašovací tabulky nebo aktualizaci záznamu, pokud záznam existuje. Druhé vlákno má za úkol exportovat záznamy dle zadaného typu timeoutu a hodnoty parametru *-t*. Obě vlákna musí přistupovat k hašovací tabulce, tudíž je tabulka považována za kritickou sekci. Vlákna se tedy synchronizují pomocí připravených mutexů. Důležité je poznamenat, že vlákno pro sebe zamyká celou hašovací tabulku, nikoli řádek se záznamem, který zrovna potřebuje. Pravidelně tedy nastává situace, že se vlákna navzájem zbytečně blokují, aniž by vyžadovala stejný záznam. To znamená, že agregator při běžném provozu nepracuje paralelně, i když v něm figurují dvě vlákna řízení.

V rámci této diplomové práce je agregátor schopen pokrýt funkcionalitu celkem tří stádií — Grouper, Window a Aggregator. Ve stádiu Grouper se extrahují identifikátory představující agregační klíče a dosadí se k parametru agregátoru *--key*. Obdobně je implementována manipulace parametrů vztahující se k typu timeoutu resp. časovému oknu, kde zdrojem informací je stádium Window. Ve stádiu Aggregator je zobrazeno ukládání výsledku agregační funkce do proměnné, ale tato akce není nikde v modulu implementovaná. Stačí ponechat implicitní chování agregátoru, kdy ukládá výsledek agregační funkce zpět pod příslušný UniRec identifikátor. S dopadem takového jednání musí být počítáno v dalších stádiích, což je řešeno strategií aliasů, viz 5.6.

Pokud je alespoň jedno stádium (Grouper, Window, Aggregator) zahrnuto v daném pravidle odhalující nežádoucí provoz, pracuje se poté v příslušném zřetězeném zpracování záznamů síťových toků s jednou instancí agregátoru. Není nutné tedy v pravidle použít

všechna tři stádia. Jestliže např. v pravidle není přítomné stádium Grouper, tak je se všemi příchozími záznamy manipulováno, jakoby měly stejný agregační klíč. Pokud chybí stádium Window, nastaví se typ časového okna jako Tumbling window s maximální možnou velikostí. A jestliže chybí stádium Aggregator, dochází pouze k agregování na základě klíče.

V rámci této diplomové práce bylo také potřeba přepsat agregátor tak, aby podporoval rozhraní znázorněné na obrázku 5.3. Díky tomuto rozhraní může být agregátor součástí zřetězeného zpracování.

5.6 Substituce proměnných

Nutnost používání aliasů vyplývá z principu fungování agregátoru. Pokud je např. zvolena agregační funkce součtu, kde parametrem pro tuto funkci je počet bytů — `SUM(BYTES)`, tak výsledek součtu se v exportovaném záznamu nachází pod identifikátorem `BYTES`. Návržený jazyk modulu Policer, ale nechá uživatele si jméno identifikátoru zvolit (např. `sumBytes`) a dále se s identifikátorem pracuje ve stádiích Group-filter a Selector. Bylo by možné agregátor upravit tak, aby výsledek zmiňovaného součtu ukládal pod identifikátor `sumBytes`, ale nastal by problém u stádia Group-filter. Filtr je implementován tak, že programátor předem připraví callback funkci definující vazbu řetězce identifikátoru s příslušným datovým typem. Tato vazba je tedy vytvořená předem staticky a není možné ji vytvořit dynamicky, kde by řetězec identifikátoru byl zjištěn za běhu programu. Z toho pro agregátor vyplývá, že exportovaný záznam nesmí obsahovat uživatelsky zvolený identifikátor (`sumBytes`), ale předem známý identifikátor (`BYTES`). Interně se tedy musí jméno `sumBytes` přejmenovat na `BYTES`. Jinak řečeno jméno `BYTES` je aliasem pro `sumBytes`.

Na obrázku architektury modulu Policer 5.1 byly znázorněny dvě hlavní části s označením Frontend a Backend. V rámci Frontend části se zjistí UniRec aliasy k proměnným ze stádia Aggregator. Na následující ukázce je zobrazen kód datové struktury, ve které se aliasy ukládají:

```
std::string branchPathT;
typedef struct varNameT_s {
    std::string name;
    std::string alias;
} varNameT;
std::map<branchPathT, std::vector<varNameT>>;
```

Datový typ `branchPathT` je řetězec znaků popisující cestu ke stádiu. Cesta je složená ze jmen větví oddělených znakem lomítko a slouží k identifikaci konkrétního stádia. Datová struktura `varNameT` se skládá z `name`, kde je vyhrazeno místo pro uložení jména proměnné, a z `alias` určeného pro uložení příslušného UniRec identifikátoru.

5.7 Odlišnosti od návrhu

Základní funkcionalitu návrhu se podařilo úspěšně implementovat. Pouze v jednom případě se musel navržený jazyk upravit a přizpůsobit se tak výsledné implementaci. Konkrétně se jedná o stádium Window, kterému se změnily parametry. Z původních parametrů stádia Window zůstal pouze parametr `RANGE`. Nový parametr reflektuje možné nastavení agregátoru, který byl představen v sekci 5.5. Parametrem `TYPE` lze zvolit typ timeoutu, kde je možné vybrat buď režim Aktivní, Pasivní, Globální či Mix. Parametr `RANGE` nadále před-

stavuje velikost časového okna, nebo lze v kontextu agregátoru na hodnotu nahlížet jako na zvolený časový limit.

Bohužel z časových důvodů se nepodařilo implementovat stádium Publisher. Nelze tedy pravidlem vytvořit hodnotu a sdílet ji ostatním větvím. Stádium Publisher mělo přinést charakteristiku monitorované sítě, jež by byla vyjádřena pomocí nějakých aktuálních statistických vlastností. Pokud by uživatel takové charakteristiky přesto potřeboval, může použít jiný dostupný nástroj pro analýzu monitorované sítě, zjistit si potřebné konstanty pro definování politiky sítě a poté použít modul Policer pro sledování porušení stanovené politiky. Uživatel se tedy bez této funkcionality může obejít a nejedná se o klíčovou část. Dále není ve stádiu Selector implementovaná možnost aritmetické úpravy nějaké hodnoty. To by ovšem také nemělo uživatele nijak omezovat, protože se stejně předpokládá, že by tato možnost byla nejvíce využívána pouze s kombinací stádia Publisher, které tedy chybí.

Poslední věcí je absence konfiguračního souboru, ve kterém měly být konfigurační proměnné, na které by se odkazovalo v pravidlech pomocí symbolu $. Na modul Policer se tedy zatím dá pohlížet jako na první prototyp tohoto konceptu, na kterém je možné demonstrovat možné splnění potenciálních potřeb správce pomocí detekčních pravidel sepsaných v jednoduchém a přehledném jazyce. Právě forma pravidel a jejich možné další rozšiřování o nové funkcionality může přinést zajímavé řešení monitorovacích úloh. Díky funkční implementaci lze také s tímto konceptem dále experimentovat.

Kapitola 6

Testování a měření výkonu

Tato kapitola se v první sekci 6.1 zabývá postupem při testování modulu Policer. Součástí této sekce je také rozbor navrženého jazyka z pohledu možných posloupností stádií, což je užitečný podklad pro kontrolování výstupů. Následující sekce 6.2 se zabývá výkonnostním měřením. Zkoumají se zde výkonnostní rozdíly různých zápisů detekčních pravidel a také souvislosti s komponentami filtr a agregátor. Kapitola končí kritickým zhodnocením modulu Policer z pohledu naměřených výsledků.

6.1 Testování

Během vývoje modulu Policer se průběžně zkoušelo správné fungování nově implementovaných částí. Na začátku se kladla pozornost především na správnou činnost sémantických kontrol a poté na bezchybnou aplikaci aliasů. Základní komponenty filtr a agregátor nebylo nutné tolik testovat, protože jejich implementace byly převzaty z prací, ve kterých testování provedli samotní autoři. Třetí a poslední základní komponenta Selector však musela být v rámci této práce prověřována. Výhodou u testování této komponenty bylo, že není zdaleka tolik složitá jako zmiňované dvě komponenty a navíc lze chyby snadno odhalit na výstupu modulu.

Potenciální nebezpečí vzniku chyby dále hrozí u sestavování parametrů pro komponentu. Jak bylo ale zobrazeno na obrázku 5.3, vstupní parametr se do komponenty vkládá jako řetězec znaků, takže ho lze snadno logovat a kontrolovat tak např. správné přejmenování identifikátorů pomocí aliasů.

Po ověření správného fungování jednotlivých komponent přišla na řadu kontrola různých variant posloupností stádií. Pro toto testování bylo naprogramováno několik pomocných skriptů v jazyce Python a Bash[1]. Skripty především pomáhají automaticky generovat detekční pravidla na základě zadané jednoduché notace. Například posloupnost stádií Filter, Aggregator, Selector se zapíše jako `fi-ag-se`. Také lze zadat, před kterým stádiem má dojít k rozvětvení zpracování. Například pokud se skriptu `generate_src.py` zadá notace `fi-^ag-se`, tak se vygenerují testovací pravidla s větvením, které je znázorněné ve výpisu 6.1. V tomto případě se také ještě zadal počet pravidel s hodnotou dvě, takže výsledkem jsou dvě pravidla `A.B` a `A.C`. Tímto způsobem lze automatizovaně testovat větvení před jakýmkoliv stádiem. Jak je z ukázky patrné, vygenerovaná pravidla nemají sice praktické využití, ale pro testování posloupností stádií postačují.

[1] `generate_src.py`, `settings.py`, `try.sh`, `scriptLib.sh`

```
branch A {
    filter: PACKETS > 0;
    branch B {
        aggregator: sum = SUM(PACKETS);
        selector: sum;
    }
    branch C {
        aggregator: sum = SUM(PACKETS);
        selector: sum;
    }
}
```

Výpis 6.1: Příklad generovaného testovacího pravidla `fi-^ag-se`.

Pomocí zmiňované notace lze vyjmenovat všechny validní varianty posloupností stádií, ze kterých lze pravidlo vytvořit. Některé posloupnosti vykazují podobné vlastnosti a lze je tedy seskupit do tříd. Následující přehled informuje o tom, ze kterých posloupností se třídy skládají a jaká je jejich charakteristika:

Tabulka 6.1: Varianty povolených posloupností stádií.

Třída	Posloupnosti							
A	f-g-w-a-gf-s	g-w-a-gf-s	f-g-w-a-s	g-w-a-s	f-s	s		
B	f-w-a-gf-s	w-a-gf-s	f-w-a-s	w-a-s				
C	f-g-a-gf-s	g-a-gf-s	f-a-gf-s	a-gf-s	f-g-a-s	g-a-s	f-a-s	a-s
D	f-g-w-gf-s	g-w-gf-s	f-g-gf-s	g-gf-s	f-g-w-s	g-w-s	f-g-s	g-s

Filter (f), Grouper (g), Window (w), Aggregator (a), Group-filter (gf), Selector (s)

Třída A představuje uskupení pravidel, jejichž posloupnosti jsou typické pro detekování nežádoucího provozu. Parametry pro agregátor jsou úplné, tzn. stádia Grouper, Window a Aggregator jsou vždy přítomné.

Třída B je charakteristická absencí stádia Grouper. Bez definování agregačního klíče se všechny příchozí data agregují do jednoho záznamu. Záznam se vlivem časového okna objeví na výstupu za nějaký stanovený čas a toto se opakovaně děje po celou dobu běhu programu.

Třída C obsahuje posloupnosti stádií, kde se nevyskytuje stádium Window. Bez definovaného časového okna se výstupní záznamy objeví až při ukončení činnosti modulu. S tím je zároveň spojeno nebezpečí, že u čítačů může dojít k přetečení, což by znamenalo nesprávné hodnoty na výstupu.

Třída D seskupuje posloupnosti, u nichž chybí stádium Aggregator. Výstupem těchto pravidel jsou záznamy agregované pouze pomocí agregačních klíčů. Implicitně ovšem záznamy obsahují položky `TIME_FIRST`, `TIME_LAST` a `COUNT`, které lze využít ve filtrovacím výrazu u stádia Group-filter.

Z popisu tříd vyplývá, že rozdíly mezi třídami jsou založeny na absenci některého stádia z trojice Grouper, Window, Aggregator. Parametry vztahující se k agregátoru nejvíce ovlivňují výstupní data a s tímto poznatkem je třeba při návrhu testovacích sad počítat.

Činnosti spojené s průběžnými kontrolami během vývoje modulu, a také díky systematické tvorbě posloupností stádií, nakonec odhalily několik chyb v implementaci, které byly následně opraveny. Implementoval jsem také pomocné skripty, které mohou být využity pro robustnější přístup při automatickém testování. V budoucnu by bylo určitě vhodné podrobit implementaci jednotkovému testování (Unit testing), potom co by se implementovaly chybějící funkce a nová rozšíření.

6.2 Měření výkonu

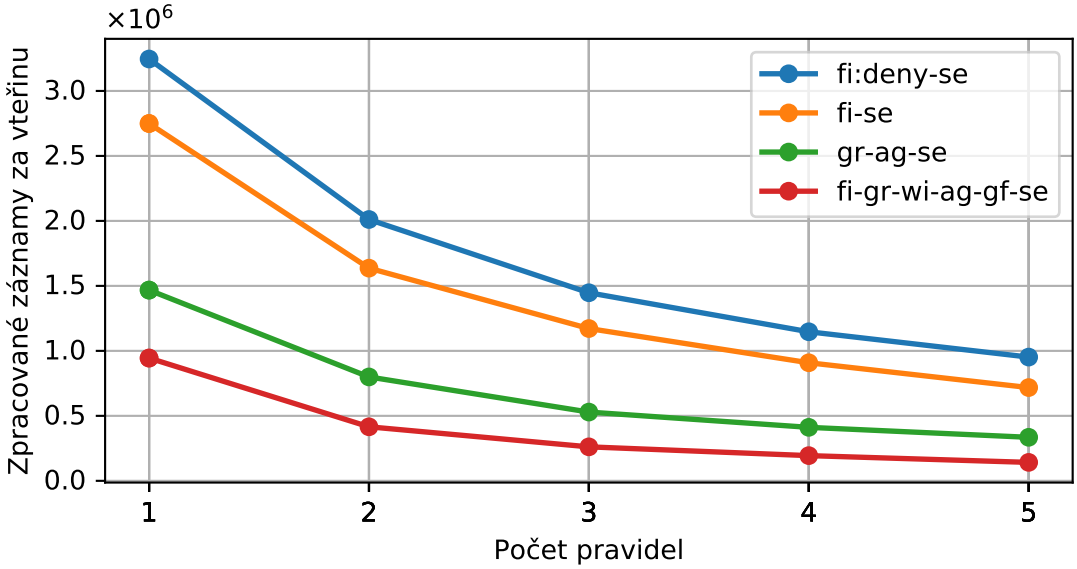
Veškerá výkonnostní měření byla realizována na osobním počítači s 4-jádrovým procesorem AMD A8-4500M pracující na frekvenci 1.9 GHz a operační pamětí DDR3 o velikosti 8 GB. Vstupní data byla získána odchycením z reálného síťového provozu sítě CESNET2 ve formě síťových toků. Data byla uložena do binárního souboru ve formátu `.trapcap` o objemu 1,2 GB. V souboru se celkově nachází 15 644 485 síťových toků. Modul Policer byl zkompilován pomocí `g++` s optimalizací `-O3`. Díky rozhraní LibTrap knihovny modul přímo načítal data z `.trapcap` souboru a výstup byl směrován do tzv. blackhole, tzn. že výstupní data jsou zahazována. Pro potřeby měření byl implementován skript `perf.sh`.

```
filter: DST_IP in [128.0.0.0/1 0.0.0.0/1] and PACKETS > 0;
grouper: SRC_IP;
window: type = global, range = 1 second;
aggregator: sum = SUM(PACKETS);
group-filter: sum > 0;
selector: SRC_IP, sum;
```

Výpis 6.2: Jednoduché parametry způsobující slabé zatížení.

V rámci měření byly vytvořeny dvě skupiny parametrů, jenž se odlišují na základě toho, jak velké působí zatížení systému. Podoba jednoduchých parametrů způsobující slabá zatížení je znázorněna ve výpisu 6.2. Ve výpisu jde vidět, že stádium Filter učiní maximálně tři porovnání a každý záznam propustí dále k dalšímu zpracování. Časové okno je typu Tumbling window o velikosti jedné sekundy, což je nejmenší možná nastavitelná hodnota. Dále se vypočítá suma paketů na základě agregačního klíče v podobě zdrojové IP adresy. Výsledek agregační funkce se vždy propustí dále přes stádium Group-filter díky pravdivému vyhodnocení podmínky a Selector nakonec pouze předá data na výstup.

Na grafu 6.1 je znázorněno měření zaměřující se na vytížení jednotlivých stádií obsahujících jednoduché parametry. Z legendy grafu lze vyčíst, že všechna zřetězená zpracování byla ve tvaru, jenž je naznačen ve výpisu 6.4. Tento výpis v podstatě sděluje informaci, že každé pravidlo je vyjádřeno pouze jednou větví. Na grafu jsou modrou a oranžovou křivkou zobrazeny naměřené výkony filtru v závislosti na tom, zda filtr všechny příchozí záznamy propustí (`fi-se`) nebo nepropustí (`fi:deny-se`) dále do stádia Selector. Pro zápis `fi:deny-se` musel být parametr stádia Filter upraven podle výpisu 6.3. Z rozdílu hodnot těchto dvou křivek vyplývá, že míra odfiltrovaných záznamů může ovlivnit výkon modulu až o několik stovek tisíc zpracovaných záznamů za vteřinu. Zelená křivka představuje míru zatížení modulu v případě nasazení agregátoru bez podpory filtrů. Z grafu je patrná mnohem větší výkonová náročnost agregátora oproti filtru. Červenou křivkou je zobrazeno zatížení v případě nasazení všech stádií. Křivka končí u pěti pravidel s hodnotou 140 000 zpracovaných záznamů za vteřinu.



Obrázek 6.1: Graf znázorňuje, kolik miliónů záznamů se zpracuje za vteřinu na základě počtu pravidel a volbě posloupnosti stádií. V tomto případě stádia obsahují jednoduché parametry způsobující slabé zatížení modulu.

```
filter: DST_IP in [128.0.0.0/1 0.0.0.0/1] and PACKETS == 0;
```

Výpis 6.3: Parametr způsobí odfiltrování každého záznamu (fi:deny).

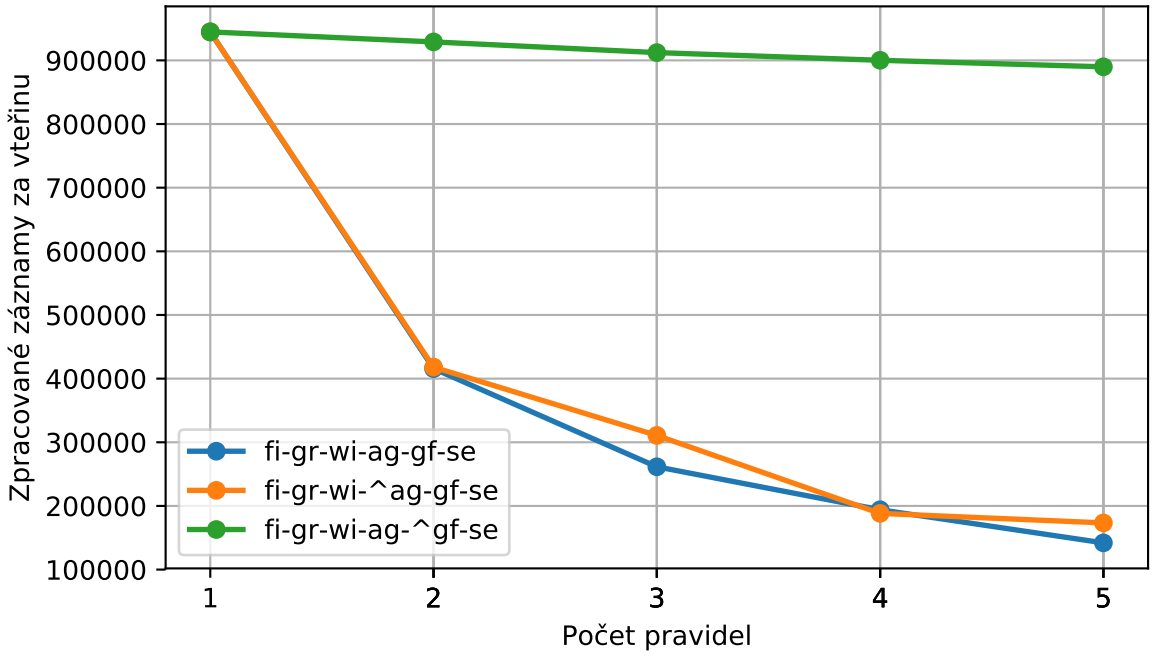
```
branch A {filter: ...; grouper: ...; ...... selector: ...;}
branch B {filter: ...; grouper: ...; ...... selector: ...;}
branch C {filter: ...; grouper: ...; ...... selector: ...;}
```

Výpis 6.4: Příklad tří pravidel tvaru `fi-gr-wi-ag-gf-se`.

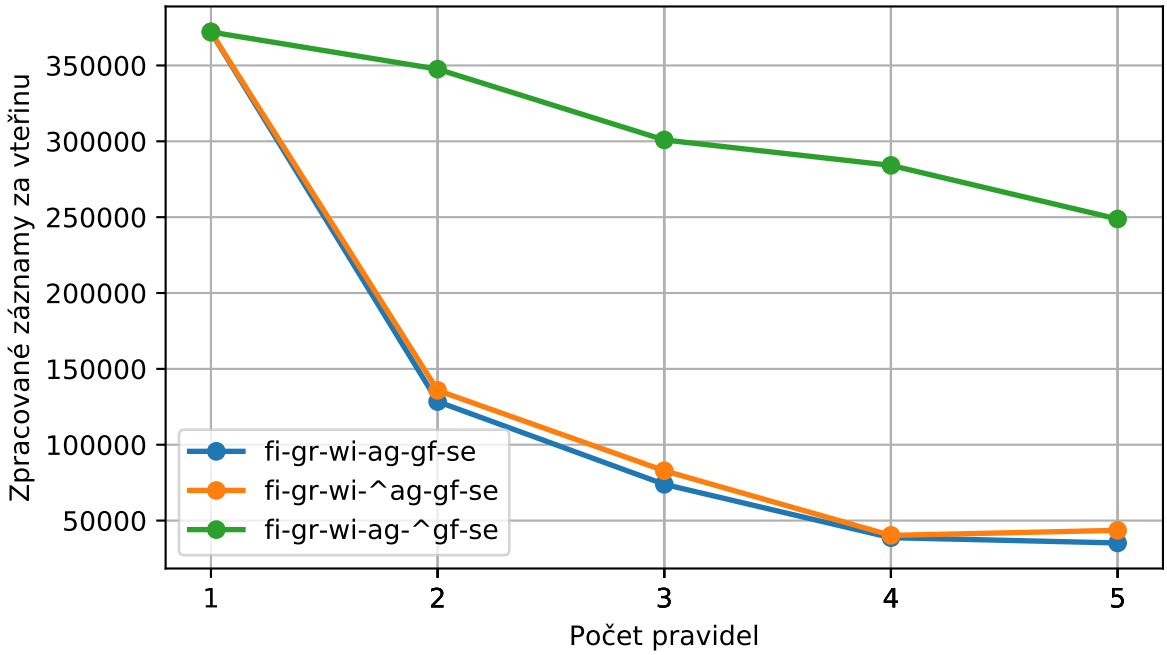
```
branch A {
    filter: ...; grouper: ...; window: ...; aggregator: ...;
    branch B {group-filter: ...; selector: ...;}
    branch C {group-filter: ...; selector: ...;}
    branch D {group-filter: ...; selector: ...;}
}
```

Výpis 6.5: Příklad tří pravidel tvaru `fi-gr-wi-ag-^gf-se`.

Na grafu 6.2 je ještě jednou zkoumán vliv jednoduchých parametrů, ale tentokrát s jinými tvary pravidel. V legendě jsou notace poznamenány symbolem ^ a byly již zmíněny v sekci 6.1. Tento symbol označuje, před kterým místem má dojít k rozvětvení zřetězeného zpracování. Tvar zápisu pro rozvětvení je demonstrován ve výpisu 6.5, kde je zobrazeno sdílení stádií Filter až Aggregator pro větve B, C a D. Měření na grafu 6.2 vypovídá, před kterým stádiem má rozvětvení největší efekt. Zelená křivka dosahuje na grafu nejlepšího výsledku, protože k rozvětvení dochází za agregátorem, tudíž se v tomto případě použije v celém řetězeném zpracování pouze jeden agregátor. Použití jen jednoho agregátoru má pozitivní dopad, protože agregátor zatěžuje systém nejvíce, jak již bylo ukázáno v před-



Obrázek 6.2: Výkon při slabém zatížení jednoduchými parametry.



Obrázek 6.3: Výkon při velkém zatížení složitými parametry.

chozím grafu 6.1. Podobný průběh oranžové a modré křivky ukazuje, že rozvětvení před agregátorem má pouze malý dopad. Sdílením stádia Filter se zlepšil výkon přibližně do deseti tisíc zpracovaných záznamů za sekundu.

Druhá skupina parametrů sepsaná ve výpisu 6.6 je schopná způsobit daleko větší zatížení modulu. Jak je z výpisu patrné, složitější parametry byly v principu sestaveny obdobně jako jednoduché parametry, ale je zde kladen větší důraz na zatížení stádií. Ve stádiu Filter lze vidět až sedm porovnání a ve stádiu Grouper je o jeden agregační klíč navíc. Časové okno je pětinásobně větší než u předešlého stádia Window. Stádium Aggregator je nyní navíc zatíženo agregační funkcí `COUNT_DISTINCT` a `MAX`. Group-filter je podobně zatížen jako stádium Filter, jelikož také musí provést sedm porovnání. Poslední Selector má za úkol předat celý záznam na výstup, ale ještě předtím přejmenovat identifikátory `sum` a `cd`.

```
filter: SRC_IP in [128.0.0.0/1 0.0.0.0/1] and TTL > 0 and
    DST_IP in [128.0.0.0/1 0.0.0.0/1] and BYTES > 0 and PACKETS > 0;
grouper: SRC_IP, DST_IP;
window: type = global, range = 5 seconds;
aggregator: sum = SUM(PACKETS), cd = COUNT_DISTINCT(SRC_PORT),
    max = MAX(BYTES);
group-filter: cd > 0 and SRC_IP in [128.0.0.0/1 0.0.0.0/1] and
    DST_IP in [128.0.0.0/1 0.0.0.0/1] and sum > 0 and max > 0;
selector: SRC_IP, DST_IP, sum_packets = sum, max, cd_src_port = cd;
```

Výpis 6.6: Složité parametry způsobující velkou zátěž

Na grafu 6.3 je znázorněno měření zaměřující se na vytížení jednotlivých stádií obsahující složité parametry. Už při použití jediného pravidla je možné vidět, že se výkon modulu vlivem složitějších parametrů snížil o 40 %. Celkově je možné vidět, že se naměřené výsledky pohybují v nižších hodnotách, než jak je tomu v grafu 6.2. Rozvětvení za agregátorem (zelená křivka) vede opět k menšímu vytížení modulu, ale lze s přibývajícími pravidly spatřit daleko větší propady. Nejhorší výsledek patří podle očekávání modré křivce, která u pěti pravidel skončila na hodnotě 35 000 zpracovaných záznamů za vteřinu.

Naměřené výsledky přinesly několik poznatků o implementaci modulu Policer. Především velmi záleží na tom, jak moc komplikované parametry jsou ke stádiím přidány. Tento faktor může způsobit zhoršení výkonu modulu i více než o polovinu. Výkon v tomto případě se o to více zhoršuje s přibývajícím počtem pravidel. Další problém byl zjištěn u agregátoru. Ten sám o sobě nevykazuje vůbec špatné výsledky, ale pokud se vyskytuje ve zřetězeném zpracování vícekrát, tak může dojít ke zhoršení výkonu až o několik stovek tisíc zpracovaných záznamů za vteřinu. Naopak u filtru se ukázalo, že nezatěžuje zpracování tolik jako agregátor. Na druhou stranu nebyly použity rozsáhlé filtrační výrazy obsahující např. velké množství IP adres. Větvení se ukázalo nejefektivnější za stádiem Aggregator, protože s přibývajícím počtem pravidel roste pouze počet filtrů. Při monitorování sítě za účelem dohledu na dodržování stanovených politik, by byla jistě potřeba mít pravidel co nejvíce. Měření se ale zabývalo maximálně pěti pravidly a i na takovém malém počtu lze vidět výkonnostní problémy. I když na druhou stranu bylo měření prováděno na slabém hardwarovém zařízení, přesto by se neefektivita projevila i na výkonném serveru. Celkově výsledky měření ukázaly, že současná implementace modulu Policer má své nedostatky, které jsou popsány výše. Pokud si jich ale bude uživatel vědom, může modul reálně používat. Výsledky měření také naznačují, že budoucí optimalizace by měly začít nejprve u agregátoru.

Kapitola 7

Závěr

Cílem této diplomové práce bylo nalézt řešení umožňující správci kontrolovat politiky a odhalovat nežádoucí provoz v lokální síti. Cíl jsem se snažil splnit návrhem a realizací jazyka, který se v implementovaném modulu systému NEMEA používá ve formě rozhraní. Vytvořený modul Policer je prototyp, jenž zatím není optimalizovaný na výkon, ale jeho potenciál je využitelný pro další vývoj, jehož cílem by bylo široké uplatnění v praxi.

Podkladem pro splnění zadání práce sloužilo v prvé řadě studium monitorovacích prvků jako je sonda, IPFIX kolektor a analytický systém NEMEA. V práci se nachází souhrn způsobů, jak se data ve formě síťových toků získají, jak probíhá jejich zpracování a na jakých základech systém NEMEA funguje. Druhá polovina teoretické části se zaměřuje na problémy doprovázející proudové zpracování. Podrobně jsou rozebrány posuvná okna, proudové systémy a jejich zajímavé funkce a jazyky.

Konkrétní nastudované systémy jsem poté podrobil rozboru v souvislosti s obecnými potřebami správce. Došel jsem k závěru, že pro tuto diplomovou práci není vhodné použít jazyk SQL, který je u proudových systémů sice typický, ale zbytečně komplexní. Hlavní inspirací této práce se místo toho stal jazyk NFQL, který svým konceptem více uspokojoval odvozené požadavky. Tento jazyk byl ale podroben značným úpravám, aby byl co nejjednodušší a hlavně aby obsahoval vyjadřovací prostředky pro splnění stanovených potřeb správce. Nejdůležitějším bodem této práce je zjištění, že s pomocí filtrování, agregace a druhého filtrování lze pokrýt velkou část úloh. O tento poznatek se opírá výsledný navržený jazyk, jenž jsem dále doplnil různými podpůrnými konstrukcemi jako např. posuvná okna, globální proměnné sdílené mezi větvemi nebo aritmetická úprava agregačních hodnot. Nejvýznamnější jazykový prostředek, který jsem navrhl, je organizace pravidel do větví, což přispívá k lepší správě jednotlivých pravidel. Navíc lze také pouhým přeuspořádáním větví dosáhnout jisté míry optimalizace.

Výsledný modul Policer jsem plně integroval do systému NEMEA. Syntaktické a sémantické analýzy jazyka jsem úspěšně implementoval pomocí knihovny Boost Spirit X3. Elementární komponenty zabývající se filtrováním a agregací jsem převzal z jiných prací a přizpůsobil architektuře modulu. Přidal jsem dvě nové agregační funkce, z nichž jedna vyžadovala složitější zásah do převzaté implementace. Podařilo se také implementovat logiku zabývající se přejmenováváním identifikátorů. Dále jsem implementoval základní komponentu zaobírající se formou výstupu. Implementovaný modul Policer nakonec nepodporuje všechny navržené jazykové konstrukce. Z časových důvodů jsem se nezabýval implementací zabývající se sdílením hodnot napříč větvemi, chybí podpora aritmetických operací či pokročilá správa pomocí konfiguračního souboru. Tyto chybějící funkce byly ovšem pouze doplňující, tudíž základní požadavky modul Policer splňuje.

Po testování následovalo měření nad reálnými daty. Měření potvrdilo odhady, že při zvyšování počtu pravidel razantně klesá výkon modulu. Stačilo postupně měřit prvních pět pravidel a rozsah se pohyboval od 900 000 do 150 000 zpracovaných toků za vteřinu (flows/s). Při nasazení složitějších pravidel byl systém zatížen v rozsahu od 350 000 do 35 000 flows/s. Modul Policer by se měl používat pouze v sítích, kde rychlost provozu nepřesahuje hranici, na kterou stačí. Pro důkladné kontrolování politiky sítě pomocí mnoha pravidel je potřeba se v budoucnu zaměřit na optimalizace v implementaci modulu.

Možností pro vylepšení výkonu modulu Policer vidím několik. Například komponentu zabývající se agregací by bylo vhodné upravit tak, aby hašovací tabulka nebyla jedna velká kritická sekce, ale aby za kritickou sekci byl považován každý jednotlivý záznam (řádek) tabulky. V komponentě filtr by mohlo docházet k optimalizaci dlouhých výrazů pomocí prefixového stromu (trie). Celkově je třeba nad modulem Policer vymyslet vhodnou strategii paralelismu. Nabízí se třeba možnost rozdělit pravidla několika výpočetním vláknům.

Literatura

[1] Aaby, A. A.: Compiler construction using flex and bison. *Walla Walla College*, 2003. URL www.admb-project.org/tools/flex/compiler.pdf

[2] Adaikkalavan, R.; Perez, T.: Secure shared continuous query processing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, ACM, 2011, s. 1000–1005.

[3] Arasu, A.; Babcock, B.; Babu, S.; aj.: Stream: The stanford data stream management system. In *Data Stream Management*, Springer, 2016, s. 317–336.

[4] Babcock, B.; Babu, S.; Datar, M.; aj.: Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ACM, 2002, s. 1–16.

[5] Bajpai, V.: *A Complete System Integration of the Network Flow Query Language*. Diplomová práce, Computer Networks and Distributed Systems (CNDS) research group at Jacobs University Bremen, 2012. URL http://cnds.eecs.jacobs-university.de/archive/msc-2012-vbajpai.pdf

[6] Bajpai, V.; Schönwälder, J.: Network flow query language—Design, implementation, performance, and applications. *IEEE Transactions on Network and Service Management*, ročník 14, č. 1, 2017: s. 8–21.

[7] Cejka, T.; Bartos, V.; Svepes, M.; aj.: NEMEA: A framework for network traffic analysis. In *2016 12th International Conference on Network and Service Management (CNSM)*, Oct 2016, ISSN 2165-963X, s. 195–201, doi:10.1109/CNSM.2016.7818417.

[8] Claise, B.; Trammell, B.; Aitken, P.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information.: RFC 7011. URL www.tools.ietf.org/html/rfc7011

[9] Dayarathna, M.: Apache Storm vs WSO2 Stream Processor, Part 1. 2019. URL www.dzone.com/articles/apache-storm-vs-wso2-stream-processor-a-side-by-si

[10] Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, ročník 39, ACM, 2004, s. 111–122.

[11] Guzman, J.; Kaiser, H.: Spirit 3.0.1: Boost C++ Libraries. URL www.boost.org/doc/libs/1_67_0/libs/spirit/doc/x3/html/index.html

[12] Herceg, T.: Systémy řízení proudů dat: Dotazovací jazyky I. Matematicko-fyzikální fakulta Univerzity Karlovy, 2011.
URL www.ksi.mff.cuni.cz/~pokorny/dj/prezentace/3_48.pdf

[13] Hofstede, R.; Celeda, P.; Trammell, B.; aj.: Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *Communications Surveys & Tutorials, IEEE*, ročník 16, č. 4, 2014: s. 2037–2064, ISSN 1553-877X.

[14] Huták, L.: *Nová generace IPFIX kolektoru*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2018.
URL www.fit.vutbr.cz/study/DP/DP.php?id=20939

[15] Jarosch, M.: *Klasifikace v proudu dat pomocí souboru klasifikátorů*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2013.
URL www.fit.vutbr.cz/study/DP/DP.php?id=14951

[16] Jelínek, M.: *Proudové zpracování dat v oblasti síťové bezpečnosti: Apache Spark [online]*. Diplomová práce, Masarykova univerzita, Fakulta informatiky, Brno, 2016 [cit. 2019-01-10].
URL www.theses.cz/id/cnli7r

[17] Jirsik, T.: Stream4Flow: Real-time IP flow host monitoring using Apache Spark. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2018, s. 1–2.

[18] Laštovička, M.: *Proudové zpracování dat: Bezpečnostní monitorování sítě pomocí Apache Samza [online]*. Diplomová práce, Masarykova univerzita, Fakulta informatiky, Brno, 2016 [cit. 2019-01-10].
URL www.theses.cz/id/38rd80

[19] McCready, M.; Kinsman, M.: Windowing functions (Azure Stream Analytics): Stream Analytics Query Language Reference. 2019.
URL https://docs.microsoft.com/en-us/stream-analytics-query/windowing-azure-stream-analytics

[20] Quick, M.; Hollowood, E.; Miles, C.; aj.: World's Biggest Data Breaches and Hacks.
URL www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/

[21] Quittek, J.; Zseby, T.; Claise, B.: Requirements for IP Flow Information Export (IPFIX): RFC 3917.
URL www.tools.ietf.org/html/rfc3917

[22] Slabihoudek, M.: *Univerzální agregační modul pro systém NEMEA*. Diplomová práce, České vysoké učení technické v Praze. Výpočetní a informační centrum., 2018.
URL http://hdl.handle.net/10467/76360

[23] Štoffa, I.: *Filtr IP toků*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017.
URL www.fit.vutbr.cz/study/DP/BP.php?id=20008

[24] Suhothayan, S.: Making Real-Time Applications Simpler with WSO2 Stream Processor. 2018.
URL www.wso2.com/library/article/2018/01/making-real-time-applications-simpler-with-wso2-stream-processor/

[25] Sýkora, P.: *Dolování v proudu dat*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2009.
URL www.fit.vutbr.cz/study/DP/DP.php?id=5440

[26] Søberg, J.: *Design, Implementation, and Evaluation of Network Monitoring Tasks with the TelegraphCQ Data Stream Management System*. Diplomová práce, University of Oslo, 2006.
URL http://hdl.handle.net/10852/9436

[27] Thakkar, H.; Zaniolo, C.: Introducing Stream Mill: User-Guide to the Data Stream Management System, its Expressive Stream Language ESL, and the Data Stream Mining Workbench SMM. *Computer Science Department, UCLA*, 2010.

[28] Trammell, B.; Boschi, E.: Bidirectional Flow Export Using IP Flow Information Export (IPFIX).: RFC 5103.
URL www.tools.ietf.org/html/rfc5103

[29] Wikipedia: Síťové zabezpečení — Wikipedia, The Free Encyclopedia. 2019, [Online; accessed 16-January-2019].
URL https://cs.wikipedia.org/wiki/S%C3%AD%C5%A5ov%C3%A9_zabezpe%C4%8Den%C3%AD

[30] Wirth, N.: Extended backus-naur form (ebnf). *Iso/Iec*, ročník 14977, 1996: str. 2996.

[31] Wrona, J.: *Optimalizace distribuovaného kolektoru síťových toků*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.
URL www.fit.vutbr.cz/study/DP/DP.php?id=18670

[32] Zendulka, J.: Získávání znalostí z netradičních typů dat: Dolování v proudech dat. FIT VUT v Brně, 2017.

Příloha A

Gramatika navrženého jazyka

Následuje ukázka necelé gramatiky upraveného jazyka NFQL. Gramatika je ve formátu PEG[1].

```
#------------------
# symbol tables
#------------------
unirec_default <-
"SRC_IP" / "DST_IP" / "BYTES" / "LINK_BIT_FIELD" /
"TIME_FIRST" / "TIME_LAST" / "PACKETS" / "SRC_PORT" / "DST_PORT" /
"DIR_BIT_FIEL" / "PROTOCOL" / "TCP_FLAGS" / "TOS" / "TTL"

aggr_func_with_param_keyword <-
"COUNT" / "SUM" / "AVG" / "MIN" / "MAX" / "FIRST" / "LAST" / "OR" / "AND"
aggr_func_without_param_keyword <- "COUNT" / "BPS"

grouper_case_keyword <- "_bidirectional_"

sel_additive_op <- "+" / "-"
sel_multiplicative_op <- "*" / "/"
sel_unary_op <- "+" / "-"

#------------------
# other nonterminals
#------------------
identifier <- [a-zA-Z_] [a-zA-Z_0-9]*
#tcp_flags <- ...
#ip_addr <- ...
#mac_addr <- ...
#double <- ...

#------------------
# main divide
#------------------
main_branches <- main_branch main_branch*
```

[1] http://bford.info/packrat/

```
main_branch <- "branch" identifier '{' main_branch_body '}'
main_branch_body <- stage / main_branch+

stage <-
filter_stage / grouper_stage / window_stage /
aggregator_stage / selector_stage

#------------------
# filter stage
#------------------
filter_stage <- "filter" ':' filter_body ';' filter_successor
filter_successor <- fi_stage_succ / branch_after_fi_s
fi_stage_succ <-
grouper_stage / window_stage / aggregator_stage / selector_stage
branch_after_fi_s <- branch_after_fi branch_after_fi+
branch_after_fi <- "branch" identifier '{' branch_after_fi_succ '}'
branch_after_fi_succ <- branch_after_fi / fi_stage_succ

# filter body
filter_body <- libnf_ffilter
# prevzato z bakalarske prace o libnf_ffiltru.
libnf_ffilter <- ff_expression
ff_expression <- '(' ff_expr ')' /
                 'not' ff_expr /
                 ff_expr 'and' ff_expr /
                 ff_expr 'or' ff_expr
ff_expr <- 'any' /
           'exist' ff_field
           ff_field 'in' '[' ff_value_list ']' /
           ff_field ff_operator ff_value /
ff_field <- direction_keyword input_keyword / input_keyword
ff_operator <- '<' / 'LT' /
               '>' / 'GT' /
               '=' / '==' / 'EQ' /
               '&' / eps
ff_value <- [0-9]+ / identifier / tcp_flags / ip_addr / mac_addr

#------------------
# grouper stage
#------------------
grouper_stage <- "grouper" ':' grouper_body ';' grouper_successor
grouper_successor <- gr_stage_succ / branch_after_gr_s
gr_stage_succ <- window_stage / aggregator_stage
branch_after_gr_s <- branch_after_gr branch_after_gr+
branch_after_gr <- "branch" identifier '{' branch_after_gr_succ '}'
branch_after_gr_succ <- branch_after_gr / gr_stage_succ

# grouper body
```

```
grouper_body <- grouper_case_keyword / grouper_keys
grouper_keys <- input_keyword / (',' input_keyword)*

#------------------
# window stage
#------------------
window_stage <- "window" ':' window_body ';' window_successor
window_successor <- aggregator_stage / branch_after_wi_s
branch_after_wi_s <- branch_after_wi branch_after_wi+
branch_after_wi <- "branch" identifier '{' branch_after_wi_succ '}'
branch_after_wi_succ <- branch_after_wi / aggregator_stage

# window body
window_body <- window_wattr ',' window_range (',' window_slide)?
window_wattr <- ("wattr" / "WATTR") '=' input_keyword
window_range <- ("range" / "RANGE") '=' window_seconds
window_slide <- ("slide" / "SLIDE") '=' window_seconds
window_seconds <- ([0-9]+ ("seconds" / "second")) /
                  ([0-9]+ ("minutes" / "minute")) /
                  ([0-9]+ ("hours" / "hour"))

#------------------
# aggregator stage
#------------------
aggregator_stage <-
"aggregator" ':' aggregator_body ';' aggregator_successor
aggregator_successor <- ag_stage_succ / branch_after_ag_s
ag_stage_succ <- group_filter_stage / selector_stage
branch_after_ag_s <- branch_after_ag branch_after_ag+
branch_after_ag <- "branch" identifier '{' branch_after_ag_succ '}'
branch_after_ag_succ <- branch_after_ag / ag_stage_succ

# aggregator body
aggregator_body <- aggr_assignment (',' aggr_assignment)*
aggr_assignment <- identifier '=' aggr_func_type
aggr_func_type <- aggr_func_without_param / aggr_func_with_param_s
aggr_func_with_param_s <-
aggr_func_with_param_keyword '(' input_keyword ')'
aggr_func_without_param <- aggr_func_without_param_keyword '(' ['*']? ')'

#---------------------
# grouper-filter stage
#---------------------
group_filter_stage <-
"group-filter" ':' group_filter_body ';' group_filter_successor
group_filter_successor <- selector_stage / branch_after_gf_s
branch_after_gf_s <- branch_after_gf branch_after_gf+
branch_after_gf <- "branch" identifier '{' branch_after_gf_succ '}'
```

```
branch_after_gf_succ <- branch_after_gf / selector_stage

# group-filter body
group_filter_body <- libnf_ffilter

#------------------
# selector stage
#------------------
selector_stage <- "selector" ':' selector_body ';'

# selector body
selector_body <- sel_assignment (',' sel_assignment)*
sel_assignment <- sel_output ('=' sel_expression)?
sel_output <-
input_keyword / '"' (!(lit('"')) .)+ '"' / identifier

# selector arithmetic expression
sel_expression <- sel_additive_expr
sel_additive_expr <- sel_multiplicative_expr
                     (sel_additive_op sel_multiplicative_expr)*
sel_multiplicative_expr <- sel_unary_expr
                           (sel_multiplicative_op sel_unary_expr)*
sel_unary_expr <- sel_primary_expr / (sel_unary_op sel_primary_expr)
sel_primary_expr <- double / sel_variable / '(' sel_expression ')'
sel_variable <- input_keyword / identifier
```

Příloha B

Řešení potenciálních potřeb správce sítě

```
branch Detect_reflector{
  grouper: SRC_IP;
  window: type = global, range = 30 seconds;
  aggregator:
    sum_packets = SUM(PACKETS), // BYTES_PER_PACKET(*)
    sum_bytes = SUM(BYTES);
  group-filter:
    sum_packets > 10000 and sum_bytes > 1500000;
  selector: SRC_IP, TIME_FIRST, TIME_LAST;
}
```

Výpis B.1: Detekce reflektorů: žádná IP adresa nesmí odesílat příliš mnoho velkých paketů.

```
branch SIP_invite_flood{
    filter: DST_IP == 192.168.0.2 and
        (SIP_MSG_TYPE == 5 or SIP_MSG_TYPE == 1);
    grouper: SRC_IP, DST_IP;
    window: type = global, range = 30 seconds;
    aggregator: connection_count = COUNT(*);
    group-filter: connection_count > 100;
    selector: attacker = SRC_IP, victim = DST_IP;
}
```

Výpis B.2: SIP invite flood: na ústřednu nesmí jít více než X REGISTER/INVITE příkazů.

```
branch Peer2Peer{
    filter: DST_PORT > 49152 and DST_PORT < 65535;
    grouper: SRC_IP;
    window: type = global, range = 30 seconds;
    aggregator: p2p_count = COUNT_DISTINCT(DST_IP);
    group-filter: p2p_count > 20;
    selector: SRC_IP, p2p_count;
```

```
}
```

Výpis B.3: Peer2Peer: žádná IP adresa nesmí komunikovat s více než X adresami na dynamických číslech portů.

```
branch Blacklist{
  filter: DST_IP in [192.168.0.2 177.169.0.2 ...];
  selector: SRC_IP, DST_IP;
}
```

Výpis B.4: Detekce komunikace s adresami na blacklistech.

```
branch One_TCP_flag{
    filter: PROTOCOL == 6 and
        (TCP_FLAGS == 2 or TCP_FLAGS == 4 or TCP_FLAGS == 16);
    grouper: SRC_IP;
    window: type = global, range = 30 seconds;
    aggregator: count = COUNT(*);
    group-filter: count > 50;
    selector: SRC_IP, count;
}
```

Výpis B.5: Příliš mnoho toků obsahujících pouze jeden TCP flag - verze 1.

```
branch One_TCP_flag{
    filter:
        PROTOCOL == 6 and
        (TCP_FLAGS == 2 or TCP_FLAGS == 4 or TCP_FLAGS == 16);
    grouper: SRC_IP;
    window: type = global, range = 30 seconds;
    aggregator: flags = OR(TCP_FLAGS), count = COUNT(*);
    group-filter: (flags == 2 or flags == 4 or flags == 16) and
        count > 1000000;
    selector: SRC_IP;
}
```

Výpis B.6: Příliš mnoho toků obsahujících pouze jeden TCP flag - verze 2.

```
branch A{
    filter: SRC_IP in [198.51.100.0/24] and PROTOCOL == 1;
    grouper: SRC_IP;
    window: type = global, range = 30 seconds;
    aggregator: bps = RATE(BYTES);
    group-filter: bps > 2000000;
    selector: SRC_IP, bps;
}

branch B{
    filter: DST_IP in [198.51.100.0/24], PROTOCOL == 1;
    grouper: DST_IP;
    window: type = global, range = 30 seconds;
```

```
    aggregator: bps = RATE(BYTES);
    group-filter: bps > 2000000;
    selector: DST_IP, bps;
}
```

Výpis B.7: Na žádnou IP adresu z podsítě nesmí jít více než určité množství určitého (např. ICMP) provozu za sekundu (bps/fps), to stejné i pro opačný směr.

```
branch spam_detection{
    filter: SRC_IP in [198.51.100.0/24] and DST_PORT = 25;
    grouper: SRC_IP;
    window: type = global, range = 30 seconds;
    aggregator: smtp_count = COUNT_DISTINCT(DST_IP);
    group-filter: smtp_count > 20;
    selector: SRC_IP, smtp_count;
}
```

Výpis B.8: Z dané podsítě nesmí žádná IP adresa rozesílat spam.

```
branch mac_watch{
    grouper: SRC_MAC;
    window: type = global, range = 15 minutes;
    aggregator: cd = COUNT_DISTINCT(SRC_IP);
    group-filter: cd > 1;
    selector: SRC_MAC, cd;
}
```

Výpis B.9: Hlídání, které MAC adresy mění svou IP adresu.

Příloha C

Obsah CD

```
/
├── Policer /
├── README.txt/
├── scripts/
└── text/
```

Adresář `Policer` obsahuje zdrojové soubory modulu Policer.

Soubor `README.txt` obsahuje pokyny pro zprovoznění.

Adresář `scripts` obsahuje skripty pro měření a testování.

Adresář `text` obsahuje zdrojové soubory textu diplomové práce.